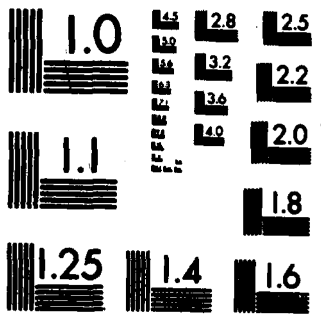

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(4)

COMPUTER SYSTEMS LABORATORY

DEPARTMENTS OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
STANFORD UNIVERSITY · STANFORD, CA 94305

VERIFICATION OF HARDWARE DESIGN CORRECTNESS: SYMBOLIC EXECUTION TECHNIQUES AND CRITERIA FOR CONSISTENCY

Warren E. Cory

Technical Report No. 83-241

June 1983

This research was supported by ARO Contract No. DAAG29-80-K-0046.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 17200.7-EL	2. GOVT ACCESSION NO. A136 406	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Verification of Hardware Design Correctness: Symbolic Execution Techniques and Criteria for Consistency		5. TYPE OF REPORT & PERIOD COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Warren E. Cory		8. CONTRACT OR GRANT NUMBER(s) DAAG29 80 K 0046
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University Stanford, CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE Jun 83
		13. NUMBER OF PAGES 217
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Symbolic executions of event-driven simulations tend to split into many cases. The substitution of non-selective trace for selective trace eliminates this case splitting for some elements. Case merging using conditional expressions IS a mor general technique for eliminating case splitting. New criteria for the compariso of two programs apply to both deterministic and nondeterministic models, with and without don't-cares. They allow hierarchical verification and facilitate the verification of programs with loops. A modification of a top-down hierarchical design methodology allows partitioned verification even when the design introduces new timing detail

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

VERIFICATION OF HARDWARE DESIGN CORRECTNESS: SYMBOLIC EXECUTION TECHNIQUES AND CRITERIA FOR CONSISTENCY

Warren E. Cory

Technical Report No. 83-241

June 1983

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

Symbolic executions of event-driven simulations tend to split into many cases. The substitution of non-selective trace for selective trace eliminates this case splitting for some elements. Case merging using conditional expressions is a more general technique for eliminating case splitting.

New criteria for the comparison of two programs apply to both deterministic and nondeterministic models, with and without don't-cares. They allow hierarchical verification and facilitate the verification of programs with loops. A modification of a top-down hierarchical design methodology allows partitioned verification even when the design introduces new timing details.

Key Words and Phrases: event-driven simulation with selective trace, non-selective trace, case analysis, case merging, graph models for behavior, hierarchical partitioned (piecewise) verification

Abstract

This thesis investigates two aspects of the functional verification problem in digital hardware design. ~~First,~~ It discusses the reduction of case splitting in the symbolic execution of event-driven simulations and ~~Second, it~~ defines criteria for consistency between two hardware descriptions.

Symbolic executions of event-driven simulations tend to split into many subcases due to the event detection mechanism. The substitution of non-selective trace for selective trace eliminates this case splitting for combinational elements. In general, an inductive assertion analysis or a symbolic execution test can determine whether the substitution is correct for a particular element. Case merging using conditional expressions is a more general technique for eliminating case splitting, but it tends to generate complex symbolic expressions. Case merging is most likely to be beneficial for functional verification when the simulation relation is sparse and the symbolic expression complexity grows slowly.

Milner and Brand defined criteria for the comparison of two programs to establish their consistency. This thesis defines ~~extends their work by defining~~ a criterion that applies to both deterministic and nondeterministic models, with and without don't-cares. Hierarchical verification is demonstrated for the new criterion. This criterion is also modified to facilitate the verification of programs with loops. It ~~The modified criterion~~ is shown to retain intuitive notions of correctness captured by the first criterion.

Finally, a modification of the hierarchical design methodology allows the application of Brand's theorem for partitioned verification even when the design introduces new timing details. The introduction of new timing details is performed separately from the introduction of new structural detail.

Acknowledgments

Willem vanCleemput, my adviser, patiently gave his support and encouragement. Early suggestions by John Hennessy substantially improved the thesis. Later, John Hennessy and Robert Gray served on the reading committee. Robert Shostak at SRI International provided a theorem prover for an experimental symbolic execution system which yielded some of the ideas in Chapters 4 and 5. Edward Rice and John Meadors encouraged and advised me in my high school and undergraduate days. Thank you, everyone.

A National Science Foundation Graduate Fellowship provided financial assistance for my first three years of graduate study. Willem vanCleemput and Robert Dutton supported me primarily by ARO Contract No. DAAG29-80-K-0046, with earlier support from JSEP Contract Nos. N-00014-85-C-0601 and DAAG29-79-C-0047 and gift funds from Tektronix, Inc.

Finally, I wish to thank my friends. You gave me your unswerving, caring support through good times and bad. And it was only by the considerable grace of God that I could aspire to undertake this task, let alone finish it.

In the beginning was the Word, and the Word was with God, and the Word was God. He was with God in the beginning.

Through him all things were made; without him nothing was made that has been made. In him was life, and that life was the light of men. The light shines in the darkness, but the darkness has not understood it.*

To Mom and Dad

*Scripture text: *The Holy Bible, New International Version*. Copyright © 1978 by New York International Bible Society. Used by permission of New York International Bible Society.

Table of Contents

List of Figures

Chapter 1
Introduction

This thesis investigates the verification of electronic digital hardware designs. The two areas of concern are the symbolic execution of hardware descriptions and criteria for establishing the correctness of hardware descriptions.

1.1. Hardware Descriptions and Simulation

Hardware descriptions are textual or graph descriptions often similar to computer programs and commonly used to describe the behavior or structure of a proposed hardware design. A verification procedure establishes that the design, expressed by a hardware description, satisfies design objectives given in the specification. The specification itself may be represented by a hardware description.

A computer simulation predicts the behavior of a hardware description for specific input data. Event-driven simulation is a particular type of simulation which represents a design as a collection of independent components with some means of communication between them. In such a simulation, a component remains inactive unless some condition arises (the "event").

Symbolic execution is a modification of conventional simulation that tests the hardware description ideally for all possible input values simultaneously. Such an execution may split into many separate cases ("case splitting").

This thesis investigates two aspects of the verification problem. First, it considers methods for reducing case splitting in the symbolic execution of event-driven simulations. Second, it discusses criteria for the comparison of two hardware descriptions.

1.2. Hardware Verification

The hardware verification problem can be divided into three areas: functional verification, performance verification, and design rule checking. The goal of functional verification is to establish that a proposed hardware design will correctly perform its intended function without regard to performance constraints such as speed, power, etc. Performance verification checks the design against these additional performance requirements. A design rule check tests the physical properties of the final physical design.

Although the boundaries between these three types of verification are not sharp, we can characterize the problem types by the inputs typically used in the verification process.

In functional verification, the inputs are usually two hardware descriptions of the hardware system to be built. Each description completely specifies the system behavior. One description is a system specification, while the other is a proposed design. The verification determines that the second description is a correct implementation of the first.

Alternatively, the input for functional verification may be a single hardware description annotated with assertions, where the assertions alone constitute a complete specification of how the hardware should behave. The verification checks that the hardware description satisfies its assertions. For example, Floyd's inductive assertion method is one technique for performing this test [Fl67, HK76, Ho69, LG79].

In performance verification, a typical input is a hardware description annotated with assertions. The assertions specify behavioral requirements, but do not completely specify the desired system behavior. For example, they may establish speed or power requirements. At a more abstract level, an assertion about a resource arbiter might require that if a system requests a resource infinitely often while the resource is available, then eventually it will be granted the resource, even in the presence of competing requests.

Design rule checking is like performance verification, except that the hardware description and assertions are structural (physical) rather than behavioral. Of course, behavioral considerations can motivate physical assertions ("design rules").

1.3. Functional Verification

This thesis considers the functional verification problem in which we compare two distinct behavioral hardware descriptions to establish that one is consistent with the other. In this thesis, we most often use the term "consistent" rather than "equivalent." The term "equivalent" suggests a condition that may be stronger than necessary for a design to be correct. A design can be correct (consistent with the original specification) without being equivalent to the original specification, due to

don't-cares, optimizations, and changes in the representation of data ("level of abstraction").

We can divide the functional verification process into two steps:

1. Analyze each description to characterize its behavior.
2. Compare the resulting behavioral characterizations.

This thesis considers each of these steps independently, as described in the following sections.

1.3.1. Analysis of a Hardware Description

One method of analyzing a hardware description is to simulate it for all possible input values, recording the results. Such exhaustive simulation is intractable for almost all descriptions. However, symbolic execution is a popular modification that, in many cases, allows the exhaustive simulation of a description with a small number of simulations [CM80, EG77, HK76, Ki76a].

Previous research with symbolic execution has not applied it to event-driven simulations. A symbolic execution of an event-driven simulation may still be intractable because the execution splits into too many cases. This thesis will define a simple model for event-driven simulation similar to that supported by SABLE [Hi80]. It then investigates ways of reducing the numbers of cases in symbolic executions of this model.

1.3.2. Comparison of Analysis Results for Two Descriptions

Given a characterization of two behavioral hardware descriptions, we need a criterion for comparing the results to establish whether the proposed implementation is consistent with the specification. Milner and Brand both suggested criteria for consistency using a graph model to abstractly represent behavior [Br78, Mi71]. This thesis considers the problem further using a similar model. The following observations apply:

Isomorphism of the graph models is too strong a condition for consistency. Due to don't-cares, optimizations, and differences in levels of abstraction, we do not expect that the graph models for the specification and the implementation will be identical.

It is possible that a criterion for consistency may not be commutative. That is, graph models *A* and *B* could be consistent assuming model *A* corresponds to the specification, while failing to be consistent if *B* is the specification. Therefore in the discussion we always identify which graph corresponds to the specification.

A desirable property of a criterion for consistency is that it should be usable in a hierarchical, partitioned fashion. The desired hierarchical property is that the criterion should be transitive. If graph *B* is consistent with graph *A*, and graph *C* is consistent with *B*, then we would like to conclude that *C* is consistent with *A*. This facilitates a hierarchical design strategy in which the final design is generated by a series of refinements.

The desired partitioned property is that the criterion should be applicable in a

piecewise fashion. Suppose graph *A* consists of two subgraphs *A1* and *A2* representing two components in the system. Suppose that graph *B* is similarly partitioned. Then to prove *B* consistent with *A*, it should suffice to demonstrate that *B1* is consistent with *A1* and *B2* is consistent with *A2*. Figure 1-1 illustrates the situation.

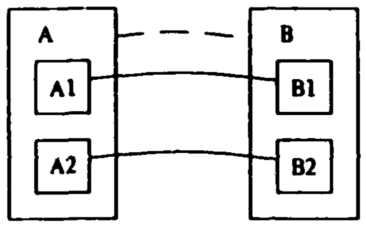


Figure 1-1: Partitioned comparison of graph models

1.4. Organization of the Thesis

Chapter 2 discusses previous work in symbolic execution, its use for hardware verification, and criteria for consistency. This chapter motivates the desirability of further work in the areas noted above.

Chapters 3 through 6 and chapters 7 and 8 form two independent sections, discussing symbolic execution and criteria for consistency, respectively. Either section may be read without consulting the other.

Chapter 3 reviews symbolic execution in greater detail and defines a model for event-driven simulation. Chapter 4 identifies simulations in which the numbers of cases in symbolic executions can be reduced by substituting non-selective trace for selective trace. Chapter 5 describes case merging, a more general technique for reducing the number of cases. Case merging tends to generate complex symbolic expressions, increasing the cost of the analysis; Chapter 6 presents two examples and considers the cost and benefits of case merging.

Chapter 7 presents a criterion for consistency applicable to both deterministic and nondeterministic models with and without don't-cares. It demonstrates the transitive property for this criterion and extends it to facilitate the verification of programs with loops. Chapter 8 reviews Brand's result for partitioned verification, noting a problem that arises when the level of abstraction is changed in a design. It then suggests a method that allows partitioned verification even when the level of abstraction changes.

Chapter 9 summarizes the entire thesis. Appendix A defines some notational conventions. Appendix B corrects a proof from Brand's report [Br78]. Appendix C presents the complete results of the symbolic execution of the first example in Chapter 6.

Chapter 2
Previous Work

Section 2.1 reviews King's work with symbolic execution. Section 2.2 discusses previous work in hardware design verification, including but not limited to the application of symbolic execution. Section 2.3 identifies an aspect of Cheatham's work that could be applied to symbolic execution. Section 2.4 reviews Milner's and Brand's criteria for consistency of programs. Section 2.5 then identifies remaining problems that are addressed in this thesis.

2.1. Symbolic Execution for Software Testing

James King developed symbolic execution, a technique for testing programs [Ki75, Ki76a]. He implemented the technique in EFFIGY, a symbolic execution system for a subset of PL/I [HC75].

At any given instant during the normal execution of a program, the program's variables have known, constant values. Arithmetic and Boolean operators will take constant operands and return constant results.

In symbolic execution, by contrast, the program's variables have symbolic values. For example, the initial value of program variable x might be denoted by the symbolic variable "%x". A symbolic variable represents a constant but

unknown value. Operators will take symbolic operands and return symbolic results. The advantage of symbolic execution is that it tests the program for all possible values of the symbolic variables, while a normal execution tests on specific (hopefully "representative") cases.

Figure 2-1 shows a simple example contrasting the two types of execution. Figure 2-1a gives three assignments which exchange the values of two distinct integer variables. Figure 2-1b shows a sample normal execution. This test verifies one of the possible executions of this code; the assignments correctly exchange the values 24 and 101.

```
x := x - y;
y := x + y;
x := y - x;
```

(a) Code to exchange two integers

```
initial              x = 24      y = 101
after x := x - y;    x = -77     y = 101
after y := x + y;    x = -77     y = 24
after x := y - x;    x = 101     y = 24
```

(b) Normal execution

```
initial              x = %x          y = %y
after x := x - y;    x = %x - %y     y = %y
after y := x + y;    x = %x - %y     y = %x
after x := y - x;    x = %y          y = %x
```

(c) Symbolic execution

Figure 2-1: Symbolic execution example

Figure 2-1c shows a symbolic execution of the three assignments. The initial values of x and y are the symbolic variables %x and %y. The final result shows that this code will exchange the values of x and y, for all possible values of %x and %y.

This basic approach is complicated by the presence of conditional branching and loops in most programs. When a conditional branch is encountered, the single symbolic execution may split into several executions, one for each path at the conditional branch, to test all possible execution sequences for the program.

Symbolic execution may also be applied to formal software verification. The standard verification technique, the inductive assertion method introduced by Floyd [Fl67, HK76, Ho69, LG79], requires the generation and proof of "verification conditions." James King discussed four methods for generating the verification conditions [Ki76b]. Two of the methods, symbolic execution and computation of weakest precondition, yield the least complex results. Symbolic execution is somewhat more cumbersome to implement than weakest precondition, but symbolic execution is attractive because of its suitability for informal testing during program development, as discussed above.

Chapter 3 reviews symbolic execution in greater detail, giving particular attention to the handling of conditional branches.

2.2. Functional Verification of Hardware Designs

2.2.1. Comparison of Specification and Design

John Darringer noted that software verification techniques should be applicable to computer hardware description languages for the functional verification of hardware designs [Da79]. Efforts have been made to apply the inductive assertion method and symbolic execution to the comparison of two hardware descriptions.

Systems developed by Patterson (STRUM, [Pa76]) and Pitchumani and Stabler [PS82] used the inductive assertion method for hardware verification. Patterson's system was tailored for the verification of microcode specified in a high-level notation. Pitchumani and Stabler's system was tailored for the verification of designs expressed at the register transfer level (RTL). In these systems, the input program represented a proposed design, while assertions embedded in this program represented the specification. The verification procedure demonstrated that the program was consistent with its assertions.

This approach uses different notations for the specification (assertions) and the implementation (ordinary program statements). By constrast, the High Level Design and Verification System (HLDVS) [EG77, OL78] designed by Ofek et al. uses a single notation for the specification and implementation. This is desirable for supporting a hierarchical design methodology. In a top-down design, for example, this would allow the implementation in one cycle of the design to become the specification in the next cycle.

The notation used by HLDVS is the non-procedural RTL language LCD (Language for Computer Design) [EG76]. An LCD file may contain several alternate descriptions (models) of a single system or component. The verification procedure guarantees that two models are equivalent. HLDVS uses symbolic execution to carry out the verification: Symbolically execute both models and compare the symbolic results.

When using the same notation for specification and implementation, a more typical approach is to completely separate the two descriptions. W.C. Carter et al. [CE77, CJ78, CJ79, JC76, LJ77] and Steve Crocker et al. [CM80, va79, vM78] each use this approach for the verification of microcode in microprogrammed machines. One description gives the specification, describing the desired architecture of the computer being designed. The second description, using the same notation, describes the proposed host machine. (The two systems also require a third description specifying the microcode; this could be considered as an initialization section for the second description.)

The separation of the two descriptions poses a new problem. In the LCD file, alternate models would manipulate the same variables, so the correspondence between the two models was self-evident. However, when the alternate models are in separate files, each has its own set of variables. This is more flexible, allowing the same data to be represented in different ways, at different levels of abstraction, in alternate models. But it also obscures the correspondence between the several models.

To solve this problem, a third input, the "simulation relation," specifies the correspondence between a specification and the implementation. Formally, a simulation relation is a set of program state pairs $\langle s_s, s_i \rangle$, where s_s (s_i) is an assignment of values to variables (including program counter and the system stack) for the specification (implementation). If $\langle s_s, s_i \rangle$ is in the simulation relation, then s_s and s_i are corresponding states.

Given a specification, proposed implementation, and simulation relation, the verification procedure then is as follows:

1. Start symbolic executions of both descriptions. The starting points are a pair of corresponding states given in the simulation relation.

2. Continue each simulation until it reaches the next state which appears in the simulation relation.
3. Check. The two descriptions should again be in a pair of corresponding states as defined in the simulation relation.
4. Repeat the above test for all pairs of states in the simulation relation.

Crocker's Microcode Verification System uses ISPS [BB79] for the descriptions of the target machine, host machine, and microcode. Carter's Microprogram Certification System uses LSS (Language for Symbolic Simulation [CE77]) for these descriptions. Carter notes that this system could be applied to different problems, e.g., the verification of machine-code implementations of algorithms.

2.2.2. Comparison of Manual and Automated Designs

An alternate verification technique is to generate a correct design from a specification automatically. The resulting design is by no means guaranteed to be optimal. However, the generated design should be functionally equivalent to a better design created manually. A proof that a manual design is equivalent to the automated design shows that the manual design is functionally identical to the original specification. J.P. Roth at the IBM T.J. Watson Research Center and N. Kawato et al. at Fujitsu Laboratories in Japan use this approach.

Roth at IBM requires that the input hardware description be a *regular algorithm* to facilitate the automatic design generation. This is guaranteed by the syntax of the input language PL/R, which is approximately a subset of PL/I. A program RTRAN translates this description to a hardware design in a particular technology [HR74].

A designer may create his or her own design which is superior to the automatically generated design. A one-to-one correspondence must exist between the inputs, outputs, and states (registers) of the two designs; i.e., the designs must be isomorphic. If this is the case, then a program VERIFY will compare the designs [Ro73, Ro75, Ro77]. VERIFY will find that the two designs are equivalent if their state transition diagrams are identical and if their outputs are identical for all states and inputs.

Under this restricted definition of equivalence, the verification procedure reduces to a comparison of acyclic combinational circuitry. This comparison is np-hard. VERIFY uses a heuristic, the Consistency or line justification step of the D-Algorithm [BF76, Ro66], to perform the comparison. The heuristic attempts to take advantage of the structure of the combinational circuitry to make the comparison more efficient.

Kawato et al. at Fujitsu extended DDL [DD75] for use in a similar verification system for synchronous designs [KS79].

2.2.3. Comparison of Manual and Automated Specifications

S. Leinwand and T. Lamdan proposed the following approach: Given an algorithmic specification and a manual design, automatically generate a second algorithmic description from the manual design and then compare the algorithmic descriptions [LL79].

The manual design contains many implementation details. In the translation of this design, these details are removed, leaving only a description of the desired

function of the circuit; hence the term "functional abstraction" used by Leinwand. He notes that if the manual design is correct, then the behavior can always be abstracted in such a way that the resulting algorithmic description is identical to the original specification.

The functional abstraction proceeds in incremental steps. Leinwand has identified at least eight levels of abstraction from final design back to function specification:

1. Final design - basic components and interconnections.
2. Same design with feedback cutpoints identified.
3. Boolean expressions for all inputs, outputs, and feedback cutpoints.
4. Guarded commands, / *trigger* / *signal <- expression*, where the *trigger* may be sensitive to signal transitions.
5. Guarded commands with signal vectors identified.
6. Nonprocedural RTL (register transfer language).
7. Procedural RTL.
8. Abstract system specification (system responses to valid inputs).

Leinwand has been successful in abstracting behavior to the level of simple RTL.

2.2.4. Conclusion

The comparison of a manual design with an automated design naturally requires the automatic generation of a correct design. This technique is heavily technology-dependent, requiring a separate compiler for each target technology. Furthermore, the approach still requires the comparison of two different descriptions. Roth and Kawato handled this problem by requiring that the manual design and automated design be isomorphic, reducing the problem to the comparison of combinational logic. This requirement prevents many optimizations, including simplifications due to don't-care conditions.

Functional abstraction requires the ability to distinguish implementation details from functional behavior, to allow the elimination of the irrelevant details. Functional abstraction also imposes a particular hierarchy of levels of abstraction.

The disadvantages of these approaches justify continued interest in the direct comparison of a proposed implementation against the specification.

2.3. Program Analysis with Conditional Expressions

Cheatham et al. sought to derive by static analysis a compact closed-form symbolic expression of the behavior of a program [CH79]. One problem is that the program may contain many alternate paths which must be combined to obtain the single result. Cheatham suggested the use of conditional expressions for this purpose.

For example, consider the statement

if *C1* **then** *X* := *C2* **else** *X* := *C3*; .

Rather than giving two sets of program results, one each for *C1 TRUE* or *FALSE*, we can generate the single conditional value

X = **if** *C1* **then** *C2* **else** *C3* .

If *X* is Boolean, this can be written

X = *C1* **and** *C2* **or not** *C1* **and** *C3* .

The generation of the conditional expressions is directed by a flow graph similar to that used by Aho and Ullman for dataflow analysis [AU77].

Although Cheatham's work was not directed toward symbolic execution or functional verification, it is mentioned here because of its applicability to symbolic execution. The use of conditional expressions will permit the combination of multiple paths that arise when a symbolic execution cannot uniquely resolve a conditional branch.

2.4. Criteria for Consistency

Of course, the comparison of two descriptions to establish their consistency requires a criterion for consistency. In Section 2.2.1, this problem was simplified because both descriptions were deterministic with no don't-care conditions. *Allowing nondeterministic descriptions, don't-cares, or loops complicates the issue.*

Milner [Mi71] and Brand [Br78] have proposed criteria for consistency between programs. Both authors use graph models to represent programs. A vertex represents a program state, while an edge represents the change from one state to another effected by execution of the program. Two graphs $P_s = \langle D_s, F_s \rangle$ and $P_i = \langle D_i, F_i \rangle$ represent the specification and implementation programs to be compared. The simulation relation is $R_{si} \subseteq D_s \times D_i$.

Milner requires P_s and P_i to be deterministic programs with no don't-cares. That is, F_s (F_i) is a function defined for all states in D_s (D_i). In this case, the criterion for consistency is

$$R_{si} F_i \subseteq F_s R_{si} \quad .$$

(Appendix A describes notational conventions for relations.) For deterministic programs with no don't-cares, this definition is equivalent to the test procedure described in Section 2.2.1.

Milner has also demonstrated the transitivity of this definition, given suitable restrictions on the simulation relation: If P_2 is consistent with P_1 and P_3 is consistent with P_2, then P_3 is consistent with P_1. This feature would be important for hardware verification in a hierarchical design, where P_2 and P_3 would represent successive refinements in the design of P_1.

Brand assumes P_s and P_i are nondeterministic models with no don't-cares. A simplified version of his criterion for consistency is

$$R_{si} F_i^* \subseteq F_s^* R_{si} \quad .$$

The presence of closure in this definition facilitates the use of symbolic execution for verification in cases where one program uses a loop while the other one does not.

Brand demonstrated how the verification of P_i against P_s by R_{si} can be partitioned. Suppose P_s consists of two concurrent processes $P_s^1 = \langle D_s, F_s^1 \rangle$ and $P_s^2 = \langle D_s, F_s^2 \rangle$: $P_s = \langle D_s, (F_s^1 \cup F_s^2) \rangle$. Similarly P_i has two concurrent processes P_i^1 and P_i^2. Brand showed that under certain restrictions, we can verify the consistency of P_i^1 with P_s^1 and the consistency of P_i^2 with P_s^2, separately; the verification of each component individually then implies the consistency of P_i with P_s.

Chapters 7 and 8 review Milner's and Brand's work in more detail.

2.5. Remaining Problems

2.5.1. Symbolic Execution

None of the previous work with symbolic execution applied it to event-driven simulation. Event-driven simulation is attractive for hardware simulation for several reasons:

- It supports the partitioning of a system into independent components which communicate with each other by means of triggering events.
- It is useful over a range of levels of abstraction, from the architectural level down to the gate level.
- It supports both synchronous and asynchronous simulation.

Event-driven simulation is not peculiar to a particular implementation, design, or language style. For example, event-driven simulations could be defined for either procedural or non-procedural languages.

An attempt to symbolically execute an event-driven simulation may lead to an excessive number of conditional branches due to the event-detection mechanism. The standard treatment of conditional branches splits a single execution into multiple executions; this results in a high number of separate executions.

It is desirable to avoid splitting a single execution into separate executions if possible. In some cases, it is correct to do this by simplifying the event-detection mechanism, substituting non-selective trace for selective trace (Chapter 4). In other cases, Cheatham's approach may be useful.

Cheatham was able to analyze programs in textual order (top to bottom), but this is not adequate for the symbolic execution of event-driven simulations. This application requires analysis of the flow graph to organize the execution of the program (Chapter 5). A second analysis uncovers loop structures which were obvious in Cheatham's language.

The use of conditional expressions is likely to increase the complexity of the symbolic expressions generated in a symbolic execution. The added cost of analyzing the more complex expressions can offset the reduction in the number of executions. A consideration of this tradeoff is desirable (Chapter 6).

2.5.2. Consistency

Milner's and Brand's criteria for consistency both apply only to program models not allowing don't-care conditions. The addition of don't-cares changes some considerations. A single criterion that applies to both deterministic and nondeterministic program models, with and without don't-cares, is desirable; properties which could be proven about this criterion would then apply to the several model types.

The presence of closure in Brand's definition facilitates verification in some cases. Unfortunately, it also allows the verification of incorrect designs. Closure must be added in a limited way to preserve intuitive notions of correctness while still allowing verification of programs with loops.

Milner's result concerning the transitivity of program consistency assumes strict partitions on the vertex (data) space and restrictions on the simulation relation. These assumptions, reasonable for the terminating programs with which Milner was

concerned, are inappropriate for programs that represent hardware specifications. A more general result is required to facilitate hierarchical verification.

Chapter 7 addresses each of these issues.

Finally, Brand's result on partitioned verification includes restrictions on accesses to variables by more than one component. This in turn implies that P_s and P_i must use similar timing protocols. This prevents partitioned verification when P_i introduces a new level of abstraction with new timing details. It is desirable to allow partitioned verification even in this case (Chapter 8).

Chapter 3
Symbolic Event-driven Simulation

Verification of design correctness by exhaustive simulation is almost always intractable, even for simple designs. Symbolic simulation addresses this problem by testing ma.., different cases simultaneously. (Symbolic simulation is the symbolic execution of a hardware description.) Each distinct case is characterized by a unique assignment of values to the symbols used in the simulation.

This chapter reviews symbolic execution. A more complete discussion was written by Hantler and King [HK76]. This chapter also describes an event-driven simulation model to which symbolic execution can be applied.

Section 3.1.1 reviews the symbolic computation of expressions. Section 3.1.2 describes case splitting and presents a suitable set of functions for implementing a case splitting execution. Section 3.2 defines a simple model representing event-driven simulation. Finally, Section 3.3 discusses the drawbacks of symbolically executing an event-driven simulation.

In this chapter and the next two chapters, we will often derive new algorithms by modifying existing algorithms. To facilitate the comparison of the new and old versions, we will adopt the convention of marking altered or new lines with vertical

bar "|" at the right margin. This bar will be preceded by the figure number (with dash), section number (with period), or page number (with "pg") against which the current figure should be compared. Sometimes, lines with small changes will not be marked to focus attention on the more *significant differences*.

In the following discussion, the term "conventional execution" will refer to an execution in which all variables have specific constant (not symbolic) values.

3.1. Review of Symbolic Execution

3.1.1. Symbolic Computations

Denote by S_c the Cartesian product of the domains of the inputs provided during the course of a given execution of a program. For example, the domain for *one scalar subrange input in a Pascal program may* be $\{0, 1, 2, 3, 4\}$. *In a* conventional execution, each input is assigned a known constant value from its domain.

At any given time during a conventional execution, the program inputs and *variables have known constant values.* The *program's* operations take constant arguments and yield constant results. Each execution tests the program only for the particular values of the program inputs supplied; only for a single value of $s \in S_c$.

The motivation for symbolic execution is that each execution should test the *program for many different values of the program inputs;* ideally for all possible values. In a symbolic execution, some or all of the inputs may be represented by symbolic variables. In our discussion, symbolic variable names will always start with "%". A symbolic variable has a constant but unknown value.

Let subset $S_s \subseteq S_c$ be the Cartesian product of the value ranges represented by the inputs in a symbolic execution. If an input x is a constant c, then the value range of x is just $\{c\}$. However, if x is assigned a symbolic variable $\%v$, then the value range of x is the entire domain of x.

In a symbolic execution, program inputs and program variables have symbolic values. That is, if a variable has domain R in a conventional execution, then its value in a symbolic execution is a function mapping S_s to R. The symbolic variables or constants assigned to the inputs are simple examples of such functions. The program's operations take symbolic arguments and yield symbolic results. A single execution tests the program for all values of $s \in S_s$.

For example, consider the statement "$x := y + z$". In a conventional execution, y and z may have values 5 and 7. The statement computes the result 12 and assigns it to x. This execution covers only the single case $y = 5$ and $z = 7$.

In a symbolic execution, by contrast, y and z may have symbolic values $\%a$ and $\%b$, where $\%a$ and $\%b$ are the symbolic inputs to the program. The statement computes the symbolic result $\%a + \%b$ and assigns it to x. The execution covers all possible values of $\%a$ and $\%b$.

Then S_s is the domain of inputs over which this execution tests the program. (Actually, this domain is constricted by a path condition, introduced below.) All symbolic values computed have domain S_s. Evaluation of the symbolic values for a particular $s \in S_s$ yields the results of a conventional execution of the program using constant inputs s.

In a discussion of symbolic execution, the reader could interpret an expression in at least two ways:

- An expression to be evaluated as in a conventional execution, yielding a particular constant value.
- An expression to be evaluated as in a symbolic execution, yielding a symbolic value.

If the expression is Boolean, a third interpretation is possible:

- An assertion about the program, true for some or all $s \in S_s$.

When the appropriate interpretation may be in doubt, we will identify expressions of the above types as being constant-valued expressions, symbolic expressions, or assertions, respectively.

In the following discussion, the first character of some symbolic execution functions and variables will be "!". This will avoid naming conflicts with user-declared identifiers.

3.1.2. Case Splitting

The fundamental problem with symbolic execution is its inability to resolve conditional branches, where a branch is taken only if a specified Boolean expression *Cond* is true. This expression has a symbolic value which may not simplify to *TRUE* or *FALSE*. In this case, the program cannot uniquely select a path to follow; for some values of $s \in S_s$, a conventional execution would follow the true branch, while for other values of s, the program would follow the false branch.

The traditional solution is to choose (interactively or automatically) either the true or false branch, maintaining a *path condition* which gives the condition under which the assumed choice was correct. When the program reaches the end of one path, we may explore the other branch separately. Once a path splits at an unresolved conditional branch, the two paths do not rejoin. The program follows each path separately until the end of execution, and then analyzes the results for each path separately. We must eventually explore all possible choices.

This technique has been called *forking* or *case analysis* [Ki76a]. The following discussion will refer to this technique as *case splitting*. This emphasizes the fact that a case analysis splits a single program execution into several executions, one for each path. In the worst case, the total number of paths in a case splitting execution may increase exponentially as the number of conditional branch points in the program, the addition of each if statement potentially doubling the total number of paths.

Case splits may arise in other situations. For example, to simplify the internal representation of data structures, a symbolic program may require that all array subscripts evaluate to constants [Ki75]. If some subscript does not evaluate to a constant, the program chooses a constant value, again updating the path condition to reflect the choice made.

We now introduce one possible set of functions to support symbolic execution with case splitting. The program uses functions *!Decide* and *!Const* to select a path or constant value, introducing a case split if necessary. Function *!Decide* examines a symbolic Boolean expression, returning *TRUE* or *FALSE*. Function *!Const* examines a symbolic scalar expression, returning a constant scalar value. The remainder of this section describes path-condition *!Pc*, *!Decide*, and *!Const*, also introducing several auxiliary functions.

1. **!Pc:** Path condition *!Pc* is a Boolean function with domain S_s. We define $!Pc(s) = TRUE$ *iff* the current symbolic execution has made the same conditional branches and assumed the same constant values as would a conventional execution with inputs s. Function *!Pc* then identifies the subset of S_s tested by this symbolic execution.

The remaining functions all take one argument $f\colon S_s \rightarrow R$, a symbolic expression, where R is any valid scalar type.

2. **!ConstP:** Predicate *!ConstP* is a Boolean function. It returns *TRUE iff* it recognizes argument f as being constant over the domain for which this symbolic execution is valid. That is, *!ConstP* returns *TRUE iff* if it can find constant $r \in R$ such that $\forall s \in S_s\colon (!Pc(s) \supset (f(s) = r))$. The program will use *!ConstP* to test the symbolic value of an expression which must evaluate to a constant, e.g. a subscript. No case split is necessary if *!ConstP* is true. Function *!ConstP* can vary in sophistication. A simple version may simply check whether f is constant. A more complex version may attempt to determine that f is equivalent to a constant, or is constant for all values of s for which $!Pc(s)$ is true.

3. **!Value:** If $!ConstP(f)$ is true, then function *!Value* returns the value $r \in R$ such that $\forall s \in S_s\colon (!Pc(s) \supset (f(s) = r))$. Otherwise, *!Value* is undefined.

4. **!Choose:** Function *!Choose* returns a value $r \in R$ such that ($!Pc$ and $(f = r)$) is satisfiable; *i.e.*, such that $\exists s \in S_s \mid (!Pc(s) \wedge (f(s) = r))$. The program calls *!Choose* when $!ConstP(f)$ is false, indicating the need for a case split. Function *!Choose* then does the following:

1. Save the current state of the program. This will allow us to return to this point later and select another path.
2. Choose the value of f to be assumed in this case. Function *!Choose* may request that the user make this choice.
3. Augment the path condition function *!Pc* to further constrain the subset of S_s for which the symbolic execution is now valid:
   a. If f is Boolean and *!Choose* returns *TRUE*, then set
      !Pc := *!Pc* **and** f .
   b. If f is Boolean and *!Choose* returns *FALSE*, then set
      !Pc := *!Pc* **and not** f .
   c. Otherwise, f is not Boolean and *!Choose* returns r; then set
      !Pc := *!Pc* **and** ($f = r$) .

5. !Const: Function *!Const* returns the constant value to be assumed for an expression, initiating a case split and altering *!Pc* if necessary. Its definition follows:

```
if !ConstP(f)
    then !Const := !Value(f)
    else !Const := !Choose(f);
```

For the remaining functions, the symbolic argument f is Boolean.

6. !IdTrue: Boolean function *!IdTrue* returns *TRUE iff* its argument f is identically true; that is, $\forall s \in S_s$: $f(s)$.

7. !Decide: Boolean function *!Decide* returns the Boolean constant value to be assumed for a conditional expression, initiating a case split and altering *!Pc* if necessary. Its definition follows:

```
if !IdTrue(!Pc ⊃ f)
    then !Decide := TRUE
    else if !IdTrue(!Pc ⊃ not f)
        then !Decide := FALSE
        else !Decide := !Choose(f);
```

This definition corresponds to the use of *!Const* with a smart version of *!ConstP*.

The appeal of symbolic execution with case splitting is its case of implementation. Given a conventional program, symbolic data types, and implementations of *!Const* and *!Decide*, the following modifications to the program allow a symbolic execution with case splitting:

1. Let user-declared variables have symbolic values rather than constant values.
2. Extend operators to allow symbolic computations as well as constant-valued computations. The form of an operation (symbolic or constant-valued) depends on the form of its operands.
3. Declare global variable *!Pc* which has a symbolic value, initially *TRUE*.
4. Replace subscripts and conditional expressions *Expr* by *!Decide*(*Expr*) or *!Const*(*Expr*).

As an example, Figure 3-1 shows the modification of a few Pascal control constructs to carry out symbolic execution with case splitting. The change is trivial for most cases. The change is nontrivial for the **for** statement because the conditional expression computation is implicit, not explicit as in the other cases.

```
Normal execution              Case splitting version

if Cond                       if !Decide(Cond)
    then Statement1               then Statement1
    else Statement2               else Statement2

case Selector of              case !Const(Selector)
    CaseBody                      CaseBody

while Cond do Statement1      while !Decide(CS!Cond)
                                  do CS!Statement1

for VarId := Lo to Hi         begin
    do Statement1             VarId := Lo;
                              !Boundn := Hi;
                              if !Decide(VarId <= !Boundn)
                                  then goto Lj
                                  else goto Lk;
                              Li: VarId := Succ(VarId);
                              Lj: Statement1;
                              if !Decide(VarId < !Boundn)
                                  then goto Li;
                              Lk: VarId := %Undefined
                              end

X := A[Expr]                  X := A[!Const(Expr)]
```

Figure 3-1: Some control constructs for case splitting execution

3.2. Model for Event-driven Simulation

The preceding section has reviewed symbolic execution for programs in general. We are concerned in particular with the symbolic execution of hardware descriptions using event-driven simulation. This section describes a model for event-driven simulations.

The model is similar to that used by SABLE [Hi80], but it is much simpler. It is roughly equivalent to a SABLE description (ADLIB description and SDL description

combined [Hi79, Hv79, va77]) with unit delays rather than variable, real (not integer) delays. Our model will avoid the SABLE concepts of component type instantiations, subprocesses, and the event queue. The model also provides no mechanisms for driving, monitoring, or halting the simulation. These features, desirable or essential in an actual implementation, would only complicate the discussion without adding any insight here.

Figure 3-2 outlines an event-driven simulator in a notation similar to Pascal [JW79]. The simulation is an infinite loop with two major steps: (1) execute the processes concurrently in the **cobegin** statement, and (2) update array *Net* from *NewNet*. Intuitively, each process corresponds to a component in a hardware system, while *Net* and *NewNet* correspond to interconnections between the components. Array *WasUpdated*, not read in Figure 3-2, is used in the next chapter.

Processes, functions, and procedures may read only *Net* and *NetChanged* of the global variables. (Hereinafter, "procedures" will include functions.) They may write only *NewNet* and *Updated*, and only by use of the **assign** statement. The statement

```
assign Expression to j
```

does the following:

```
begin NewNet[j] := Expression;
    Updated[j] := TRUE end
```

Processes and procedures may not pass globals as actual parameters to procedures.

Processes are like procedures, with the same restrictions, except for two differences. First, the process name is a number, not an alphanumeric identifier. Second, a process may use a **pause** statement, which is not available to procedures.

```
program EventDrivenSimulation;

var Net, NewNet: array [1..NumOfNets] of NetType;
    NetChanged, Updated, WasUpdated:
                        array [1..NumOfNets] of Boolean;
    i: Integer;

<function and procedure definitions>
<process definitions>

begin (* Main loop *)
initialize NewNet;
Net := NewNet;
for i := 1 to NumOfNets do NetChanged[i] := FALSE;
Updated := NetChanged;
WasUpdated := Updated;
initialize processes;
while TRUE do begin
    cobegin for i := 1 to NumOfProcs
        resume process i
        coend;
    WasUpdated := Updated;
    for i := 1 to NumOfNets do
        if Updated[i]
            then begin
                NetChanged[i] := NewNet[i] <> Net[i];
                Net[i] := NewNet[i];
                Updated[i] := FALSE
                end
            else NetChanged[i] := FALSE
    end
end.
```

Figure 3-2: Outline of event-driven simulator

The **pause** statement consists of the single keyword "**pause**". It disables (blocks) the process for the remainder of the current **cobegin** statement. A **cobegin** statement continues until all processes are blocked at **pause** statements. When the program exits the **cobegin** statement and then enters it the next time, all processes

are enabled and may continue execution from the **pause**. The end of a process definition (that is, if the process body is not an infinite loop) is equivalent to

while *TRUE* **do pause** .

To carry out an event-driven simulation, a process will typically use **pause** in a tight loop, not permitting any other action until an appropriate event occurs, as in

repeat pause until *EventOccurs* .

Figure 3-3 shows some typical uses of the pause statement in an event-driven simulation.

```
wait for the n^th cobegin from now
    for k := 1 to n do pause

wait for net j to change
    repeat pause until NetChanged[j]

wait for positive edge on net j
    repeat pause until Net[j] and NetChanged[j]

wait until condition satisfied on positive edge
    repeat pause until Net[j] and NetChanged[j] and Condition

wait until condition satisfied, but not longer than n units
    k := 0;
    repeat k := k + 1;
        pause;
        until Condition or (k ≥ n)
```

Figure 3-3: Process constructs using pause

This model is purposely designed to isolate the processes during their concurrent execution. The restrictions on their accesses to global variables imply that the computations performed by the processes in a given **cobegin** execution are independent of the interleaving of the processes' executions.

Now impose the additional restriction that associated with each element ***NewNet*[*j*]** is a single process i_j which is the only process ever permitted to write ***NewNet*[*j*]**. This implies that the final value of ***NewNet*** is also independent of the interleaving of the processes. (As a real example, the definition of SABLE specifies that the result of a simulation is undefined unless this last property holds [Hi80].)

We may then equivalently define the simulation using any interleaving we wish; in particular, an interleaving in which the process executions do not overlap. Such a definition appears in Figure 3-4.

The event-driven simulator now behaves not as a concurrent, nondeterministic program, but as a sequential, deterministic program. The symbolic execution techniques developed for sequential programs may be applied to this model for event-driven simulation.

3.3. Event-driven Simulation With Case Splitting

Case splitting tends to generate an excessive number of case splits for event-driven simulations. Typically, when a process is executed, it will not perform any computation unless the value of some input *Net*[*j*] has changed; this change in value is the driving event. For example, the case splitting version of a simple process may have the following form:

```
    while TRUE do begin
        repeat pause until !Decide(NetChanged[j]);
        DoComputation
        end
```

The determination of whether or not *Net*[*j*] has changed is a conditional branch which ***!Decide*** may not be able to resolve. Hence, each execution of a process that is waiting for a change in a net is likely to introduce a case split.

```
program EventDrivenSimulation;

var  Net, NewNet: array [1..NumOfNets] of NetType;
     NetChanged, Updated, WasUpdated: array [1..NumOfNets] of Boolean;
     i: Integer;

<function and procedure definitions>
<process definitions>

begin (* Main loop *)
initialize NewNet;
Net := NewNet;
for i := 1 to NumOfNets do NetChanged[i] := FALSE;
Updated := NetChanged;
WasUpdated := Updated;
initialize processes;
while TRUE do begin
    for i := 1 to NumOfProcs do                                    3-2 |
        cobegin resume process i coend;                            3-2 |
    WasUpdated := Updated;
    for i := 1 to NumOfNets do
        if Updated[i]
            then begin
                NetChanged[i] := NewNet[i] <> Net[i];
                Net[i] := NewNet[i];
                Updated[i] := FALSE
                end
            else NetChanged[i] := FALSE
    end
end.
```

Figure 3-4: Modified equivalent event-driven simulator

Moreover, in the worst case, the total number of paths increases as the product of the numbers of paths in each process. Consider a simulation with just two processes. Suppose that processes 1 and 2 split into *M* and *N* paths, respectively, during each execution.

After the first execution of process 1, the simulation has been split into M paths. For *each* of the M paths, the subsequent execution of process 2 will split the single path into N paths, for a grand total of MN paths after the first cycle. After i cycles, the single simulation will have split into $(MN)^i$ separate simulations.

If each process splits into only two paths, e.g., when it has the simple form shown above, then after just four cycles the simulation splits into 256 separate paths. To exhaustively test this simulation over four cycles using case splitting symbolic execution, we must save the complete state of the simulation 255 times and carry out 256 separate simulations.

Such performance makes symbolic execution with case splitting unsatisfactory for many event-driven simulations. The following chapters will consider alternatives to case splitting for symbolic execution.

Chapter 4
Non-selective Trace

The property that a process will not perform any computations unless some net *Net*[*j*] receives a new value different from its old value is called *selective trace* [Ul65]. The selective trace often introduces an excessive number of case splits. This chapter shows an example in which case splitting can be reduced by the use of a non-selective trace. If a process uses non-selective trace, it will perform its computations whenever the net receives a new value, whether or not the value is different from the old value.

In the event-driven simulation model of Figure 3-4, the selective trace is typically implemented by

```
repeat pause until NetChanged[j] or NetChanged[k]
```

To use non-selective trace, we replace the references to *NetChanged* by references to *WasUpdated*, yielding the following:

```
repeat pause until WasUpdated[j] or WasUpdated[k]
```

Section 4.1 justifies the correctness of this approach for combinational processes.

Section 4.2 considers the possibility of applying this approach to a more general process type. A brief outline is as follows:

- Section 4.2.1 describes the general process type of interest.

- Section 4.2.2 defines terminology for the discussion.
- Section 4.2.3 introduces some restrictions and derives important properties of the general process type.
- Section 4.2.4 compares the process using selective trace against the process using non-selective and derives a test to establish whether the latter may be substituted for the former.
- Section 4.2.5 illustrates how assertions and the inductive assertion verification technique help in applying the test from Section 4.2.4.
- Finally, Section 4.2.6 defines a more restrictive general process type and shows how it might be tested without the use of auxiliary assertions.

4.1. Combinational Process

Figure 4-1 defines a NOR-gate process. *Net*[*A*] and *Net*[*B*] are the input nets; *Net*[*Y*] is the output net, written indirectly by assign. The nets are all assumed to be Boolean.

```
process 101; (* Nor *)
const A = 201;     (* indices to Net *)
      B = 202;
      Y = 203;     (* index to NewNet *)
begin
while TRUE do begin
    assign not (Net[A] or Net[B]) to Y;
    repeat pause until NetChanged[A] or NetChanged[B]
    end
end; (* 101 - Nor *)
```

Figure 4-1: NOR-gate process

The main body of the process consists of a tight infinite loop which specifies the behavior of a NOR-gate. This construction is typical of process definitions for combinational networks. Consider first a conventional simulation in which the nets are all constant-valued. The **assign** statement sets the output net *Net[Y]* to **not** (*Net[A]* **or** *Net[B]*). The **pause** loop waits for a change in the values of the inputs. As long as no change occurs, the value of *Net[Y]* does not need to be altered, and the process need not perform any computation. As soon as either *Net[A]* or *Net[B]* changes its value, the process performs a new computation, returning to the **assign** statement to update *Net[Y]*. Then the process will again wait in the **pause** loop. Recall that *Net[Y]* is driven only by this process.

To modify this process for symbolic simulation with case splitting, the operator and variable types are extended to allow symbolic computations, and the condition in the **pause** loop becomes

!Decide(*NetChanged*[*A*] **or** *NetChanged*[*B*]) .

The call to *!Decide* is required because array *NetChanged* has symbolic values. Function *!Decide* will introduce a case split if it cannot uniquely determine whether or not the values of nets *A* or *B* have changed.

This process uses selective trace. However, in combinational processes of this form, the use of a non-selective trace will still yield correct results. For example, consider an alternate conditional expression for the **pause** loop:

(*WasUpdated*[*A*] **or** *WasUpdated*[*B*]) .

This expression replaces symbolic *NetChanged* by the constant-valued *WasUpdated*, so the call to *!Decide* is not necessary. Suppose that at some point in a symbolic simulation, *Net[A]* has a stable symbolic value *%ExprA0*, while *Net[B]* has the value *%ExprB0*. Then *Net[Y]* has the symbolic value **not** (*%ExprA0* **or** *%ExprB0*)

due to a previous assignment by the process. When a new symbolic value *%ExprA1* is assigned to *Net[A]*, the process using the non-selective trace leaves the pause loop and assigns **not** (*%ExprA1* **or** *%ExprB0*) to *Net[Y]*, whether or not *%ExprA1* is equal to *%ExprA0*.

For the symbolic simulation to be correct, it must predict the correct value of *Net[Y]* for all possible conventional simulations. If for some test case, *%ExprA1* does not equal *%ExprA0*, then the symbolic simulation took the correct action for that case and the result will be correct. If for some other test case, *%ExprA1* equals *%ExprA0*, then it is also true that

$$\forall s \in S_s:$$

$$(\textbf{not}\ (\%ExprA0\ \textbf{or}\ \%ExprB0))(s) = (\textbf{not}\ (\%ExprA1\ \textbf{or}\ \%ExprB0))(s) \quad .$$

(The notation "(symbolic expression)(*s*)" denotes the constant value resulting from the evaluation of the symbolic expression for $s \in S_s$.) The symbolic simulation predicts no change in the value of *Net[Y]* for this case. This is also correct. The fact that the symbolic simulator executes an **assign** when a conventional simulator might not execute any assign does not result in any error in this example.

The non-selective trace is preferable to case splitting. With case splitting, the process must call *!Decide* to determine whether or not *%ExprA1* is different from *%ExprA0*, possibly introducing a case split where none is needed. With the non-selective trace, the simulator avoids the call to *!Decide*. This reduces the number of calls to *!IdTrue*. It also potentially reduces the total number of simulations, since a case split requires a separate simulation for each path to test all the paths.

Note that even the references to *WasUpdated* in the repeat condition are unnecessary. A weaker test is adequate. All that is required of the substitute condition *Cond* is

$\forall s \in S_s$: (*NetChanged*[*A*] or *NetChanged*[*B*])(s) $\supset$ *Cond*(s) .

It would be correct to replace the repeat statement by

repeat pause until *TRUE* ,

or just "**pause**". But this completely defeats the purpose of event-driven simulation: avoiding simulation of all logic in every time step.

Conversely, a stronger condition can be used for symbolic simulation. A possible symbolic repeat condition is

```
(WasUpdated[A] or WasUpdated[B]) and
        not !IdTrue(not (NetChanged[A] or NetChanged[B]))   .
```

Since *NetChanged*[*i*] $\supset$ *WasUpdated*[*i*] during process execution, this is equivalent to

```
not !IdTrue(not (NetChanged[A] or NetChanged[B]))   .
```

This condition uses the np-hard *!IdTrue* test, as the case splitting version does by calling *!Decide*, but it never introduces a case split. This more complex condition may be desirable for combinational logic in a feedback path; it could prevent continuous generation of symbolic expressions when no actual changes of net values are occurring.

4.2. Applicability of Non-selective Trace

The above example showed a process in which case splitting could be *completely eliminated* simply by replacing selective trace by non-selective trace, with no other changes. This section characterizes some processes for which this is possible.

4.2.1. Generalized Process of Interest

We will let process *i* be the process of interest. Assume that at some point in process *i*, we have the code in Figure 4-2a. The second pause only logically follows *S*. The process may actually branch to some other pause after *S*; for example, the one in the repeat loop. Statement *S* exits only through its bottom and does not include any pause.

```
(* Label L1 is a dummy for reference *)

repeat pause; L1: until C_sel;        repeat pause; L1: until C_non;
S;                                    S;
pause;                                pause;

          (a)                                   (b)
```

Figure 4-2: Generalized process for non-selective trace

The subscript "*sel*" denotes "selective trace"; the branch condition C_{sel} contains references to *NetChanged*. Denote by C_{non} the expression C_{sel} with all references to *NetChanged* replaced by references to *WasUpdated*. Subscript "*non*" means "non-selective trace." Figure 4-2b then shows the non-selective version of process *i*. Note that statement *S* may access *NetChanged*, and these accesses are not changed in the non-selective version. However, we will require that the branch condition C_{non} and statement *S* never introduce case splits.

The question of interest is this: Under what circumstances are the code segments in parts (a) and (b) of Figure 4-2 equivalent? If the two versions are equivalent, then we can substitute version (b) for version (a). This eliminates case splitting in this process, since C_{non} and *S* never introduce case splits.

4.2.2. Program State

Let s denote a program state in a conventional execution. The program state includes the values of the global variables, the main program counter, the program stack, plus the following for each process j:

- program counter Lc_j for process j,
- local variables $Locs_j$,
- a flag Run_j denoting whether or not the process is blocked.

A **cobegin** sets Run_j to $TRUE$, while **pause** resets it to $FALSE$. Note the distinction between the program state space and S_s above; S_s is the space of initial and input values covered by a symbolic execution. S_s is pertinent for symbolic executions, but now we are discussing conventional executions.

We will write "$(expression)\langle s\rangle$" to denote the value of *expression* evaluated in state s.

Any program state in which the main program counter points to the top of the **while** loop in Figure 3-4 is a "start state." Inspection of Figure 3-4 shows that in every start state s reached in an actual simulation, the following assertion holds:

$$\forall j\text{: } NetChanged[j]\langle s\rangle \supset WasUpdated[j]\langle s\rangle \quad . \tag{4.1}$$

This also always holds during the **cobegin** statement.

We now define the following functions for the non-selective process in Figure 4-2b.

- $P(s)$ is the start state obtained from start state s by one pass through the main loop in Figure 3-4.

- $P_i(s)$ is the state obtained from s by action of process i in one **cobegin** statement, from one **pause** to the next. Process i is the process of interest, Figure 4-2b.
- $P_*(s)$ is the state obtained from s by the combined actions of all remaining processes.
- $F_i(s)$ is the state obtained from s by action of statement S (plus any **goto** to a **pause** following S) in Figure 4-2b.
- $U(s)$ is the state obtained from s by the loop in Figure 3-4 that updates *Net*, *NetChanged*, and *Updated*.
- "$Vari[Seq]$" denotes the sequence $s \in Seq$: $Vari[s]$.
- "$\{w_i\}$" denotes the set of indices for those nets which may be written by process i.
- "$\{r_i\}$" denotes the set of indices for those nets which may be read by process i.

Note that $P(s) = U(P_*(P_i(s))) = U(P_i(P_*(s)))$.

4.2.3. Characteristics of the Generalized Process

Following the rule from Chapter 3, neither C_{sel} nor S may access *WasUpdated*. This reflects the philosophy that process i cannot detect the updating of a net by another process when that update does not change the value of the net. Of course, C_{non} violates this restriction, but we wish Figure 4-2b to be equivalent to 4-2a, so that the resulting computation will still be independent of ***WasUpdated***, conforming to this philosophy.

The following assertion is true for all program states s and net numbers j during process execution:

$$C_{sel}\langle s\rangle(FALSE\backslash NetChanged[j]) \supset C_{sel}\langle s\rangle(TRUE\backslash NetChanged[j]) . \quad (4.2)$$

If this did not hold, then for some s and j we would have $C_{sel}\langle s\rangle(FALSE\backslash NetChanged[j])$ true while $C_{sel}\langle s\rangle(TRUE\backslash NetChanged[j])$ is false. This means that for this case, the *absence* of a change in the value of a net would be the triggering event. We will not admit this as a reasonable triggering event.

This requirement on C_{sel} implies that for all states s arising during process execution, we have

$$C_{sel}\langle s\rangle \supset C_{non}\langle s\rangle \quad . \quad (4.3)$$

To prove this, assume it were not true. Then for some s, $C_{sel}\langle s\rangle$ is true but $C_{non}\langle s\rangle$ is false. Suppose C_{sel} refers to just one element of *NetChanged*, *NetChanged*[*j*]. Then

$$C_{non} = C_{sel}(WasUpdated[j]\backslash NetChanged[j]) \quad .$$

Since $C_{non}\langle s\rangle$ is not equal to $C_{sel}\langle s\rangle$, we know $WasUpdated[j]\langle s\rangle$ is not equal to $NetChanged[j]\langle s\rangle$. Eq. (4.1) then implies $NetChanged[j]\langle s\rangle$ is false while $WasUpdated[j]\langle s\rangle$ is true. Now we have

$$TRUE = C_{sel}\langle s\rangle = C_{sel}(FALSE\backslash NetChanged[j])\langle s\rangle \text{ and}$$
$$FALSE = C_{non}\langle s\rangle = C_{sel}(TRUE\backslash NetChanged[j])\langle s\rangle \quad .$$

But this contradicts Eq. (4.2). An inductive argument proves the contradiction when C_{sel} refers to more than one element of *NetChanged*.

Finally, condition C_{sel} must have no side effects. This is because the process should not change any variables until the triggering event described by C_{sel} occurs.

4.2.4. Comparison of the Two Processes

Consider the conventional execution (with constant, not symbolic, values) of the two code segments in Figure 4-2 from the first **pause** during one **cobegin** statement. We wish the results to be the same. We must define precisely the way in which the results must be "the same." The following are true for all start states reachable in a conventional simulation as defined in Figure 3-4:

- *Updated*[*i*] is false for all *i*.
- *Net* = *NewNet*.
- Main program counter points to top of main loop.
- Program stack is empty (except for globals).
- Run_j is false for all *j*.

In addition, we have the following characteristics of the simulation:

- By definition, the results of a simulation are independent of global variables *WasUpdated* and *i*. Therefore the values of these variables in start states are immaterial.
- Of all the processes' local variables, process *i* can modify only Lc_i, Run_i, and $Locs_i$.
- Process *i* can modify only elements $Net[\{w_i\}]$ of *Net*.
- If two simulations always generate the same values of *Net*, then the values of *NetChanged* will necessarily also be identical.

Then the only program variables we need to test in a comparison of the two processes are global variables $Net[\{w_i\}]$ and local variables Lc_i and $Locs_i$.

Returning to Figure 4-2, the actions of the two processes are clearly identical if

C_{sel} and C_{non} are equal. If C_{sel} and C_{non} are not equal, then Eq. (4.3) implies that C_{sel} is false while C_{non} is true. This is the only interesting case in the comparison of the two processes: The process using selective trace returns to the first pause without modifying any variables; the other process executes S.

The following criterion for the equivalence of the two code segments in Figure 4-2 summarizes the above considerations:

$$\begin{aligned}&\forall \text{ valid start states } s \mid Lc_i\langle s\rangle = L1:\\ &\quad (C_{non}\langle s\rangle \text{ and not } C_{sel}\langle s\rangle) \supset\\ &\qquad ((Net[\{w_i\}]\langle P(s)\rangle = Net[\{w_i\}]\langle s\rangle) \text{ and}\\ &\qquad (Lc_i\langle P(s)\rangle = L1) \text{ and}\\ &\qquad (Locs_i\langle P(s)\rangle = Locs_i\langle s\rangle)) \quad . \end{aligned} \tag{4.4}$$

Now make the following observations:

1) $P(s) = U(P_*(P_i(s)))$.

2) $Net[\{w_i\}]\langle U(P_*(P_i(s)))\rangle = NewNet[\{w_i\}]\langle P_*(P_i(s))\rangle$
$= NewNet[\{w_i\}]\langle P_i(s)\rangle$.

3) $Lc_i\langle U(P_*(P_i(s)))\rangle = Lc_i\langle P_*(P_i(s))\rangle = Lc_i\langle P_i(s)\rangle$.

4) $Locs_i\langle U(P_*(P_i(s)))\rangle = Locs_i\langle P_*(P_i(s))\rangle = Locs_i\langle P_i(s)\rangle$.

5) C_{non} implies $P_i(s) = F_i(s)$.

Eq. (4.4) is then equivalent to Eq. (4.5):

$$\begin{aligned}&\forall \text{ valid start states } s \mid Lc_i\langle s\rangle = L1:\\ &\quad (C_{non}\langle s\rangle \text{ and not } C_{sel}\langle s\rangle) \supset\\ &\qquad ((NewNet[\{w_i\}]\langle F_i(s)\rangle = Net[\{w_i\}]\langle s\rangle) \text{ and}\\ &\qquad (Lc_i\langle F_i(s)\rangle = L1) \text{ and}\\ &\qquad (Locs_i\langle F_i(s)\rangle = Locs_i\langle s\rangle)) \quad . \end{aligned} \tag{4.5}$$

Now, since the action of statement S must be independent of *WasUpdated*, then Eq. (4.5) simplifies to Eq. (4.6):

$$\begin{aligned}&\forall \text{ valid start states } s \mid Lc_i\langle s\rangle = L1:\\ &\quad (\text{not } C_{sel}\langle s\rangle) \supset ((NewNet[\{w_i\}]\langle F_i(s)\rangle = Net[\{w_i\}]\langle s\rangle) \text{ and}\\ &\qquad (Lc_i\langle F_i(s)\rangle = L1) \text{ and}\\ &\qquad (Locs_i\langle F_i(s)\rangle = Locs_i\langle s\rangle)) \quad . \end{aligned} \tag{4.6}$$

Then the code segments in Figure 4-2 must have the form in Figure 4-3, where Eq. (4.6) implies (not $C_{sel}\langle s\rangle) \supset C2\langle F_i(s)\rangle$. ($S'$ plus the computation and branch on $C2$ in Figure 4-3 comprise statement S in Figure 4-2.) This implication, plus the requirement that Figure 4-3b not introduce a case split even if C_{sel} is not identically *TRUE* or *FALSE*, implies that $C2$ is *TRUE* (or may be optimized to *TRUE*). Then the final form implied for the original code segments is as shown in Figure 4-4.

```
repeat repeat pause; L1: until $C_{sel}$;
        $S'$
        until not $C2$;
pause;
```

(a)

```
repeat repeat pause; L1: until $C_{non}$;
        $S'$
        until not $C2$;
pause;
```

(b)

Figure 4-3: Modified generalized process

```
while TRUE do begin
        repeat pause; L1: until $C_{sel}$;
        $S$
        end;
```

(a)

```
while TRUE do begin
        repeat pause; L1: until $C_{non}$;
        $S$
        end;
```

(b)

Figure 4-4: Final generalized process

In conclusion, given a code segment of the form shown in Figure 4-4a satisfying the properties given in Section 4.2.3, we can replace it by the code in Figure 4-4b if Eq. (4.7) is satisfied:

$$\forall \text{ valid start states } s \mid Lc_i\langle s\rangle = LI:$$
$$(\text{not } C_{sel}\langle s\rangle) \supset ((NewNet[\{w_i\}]\langle F_i(s)\rangle = Net[\{w_i\}]\langle s\rangle) \text{ and}$$
$$(Locs_i\langle F_i(s)\rangle = Locs_i\langle s\rangle)) \quad . \qquad (4.7)$$

Eq. (4.7) is identical to Eq. (4.6) except that the reference to Lc_i is removed. This is because the control structure of the final generalized process guarantees that Lc_i will have the proper value. The new version (Figure 4-4b) introduces no case splits. The example in the next section shows that assertions may be useful for properly identifying the set of "valid start states" in Eq. (4.7).

4.2.5. Use of Assertions

Now reconsider the example in Figure 4-1. This process definition is transformed in Figure 4-5 so that it conforms to the pattern in Figure 4-2a.

```
process 101; (* Nor *)
const A = 201;     (* indices to Net *)
      B = 202;
      Y = 203;     (* index to NewNet *)
begin
assign not (Net[A] or Net[B]) to Y;
while TRUE do begin
    repeat pause; LI: until NetChanged[A] or NetChanged[B];
    assign not (Net[A] or Net[B]) to Y
    end
end; (* 101 - Nor *)
```

Figure 4-5: Modified NOR-gate process

In this example, C_{sel} is

$$(NetChanged[A] \text{ or } NetChanged[B]) \quad .$$

We wish to replace this by C_{non},

$$(WasUpdated[A] \text{ or } WasUpdated[B]) \quad .$$

To justify this, we must prove that Eq. (4.7) holds for any "valid" start state s with $Lc_i\langle s\rangle = LI$. (Note that the loop satisfies the restrictions from Section 4.2.3, including Eq. (4.2).)

Unfortunately, it is trivial to choose a start state s that does not satisfy this condition; simply set every element of arrays *Net* and *NetChanged* to *FALSE*. Then $C_{sel}\langle s\rangle$ is false, but

$$NewNet[Y]\langle F_i(s)\rangle = \text{not } (Net[A]\langle s\rangle \text{ or } Net[B]\langle s\rangle) = 1$$
$$\neq Net[Y]\langle s\rangle \quad .$$

We must add assertions to the process to constrain the set of valid start states. Figure 4-6 shows a possible solution.

```
process 101; (* Nor *)
const A = 201;     (* indices to Net *)
      B = 202;
      Y = 203;     (* index to NewNet *)
begin
assert TRUE;
assign not (Net[A] or Net[B]) to Y;
while TRUE do begin
    repeat pause;
          LI: assert (not (NetChanged[A] or NetChanged[B])) ⊃
                    (Net[Y] = not (Net[A] or Net[B]))
          until NetChanged[A] or NetChanged[B];
    assign not (Net[A] or Net[B]) to Y
    end
end; (* 101 - Nor *)
```

Figure 4-6: NOR-gate process with assertions

With assertions, the process in Figure 4-6 clearly satisfies Eq. (4.7). The proof that C_{sel} can be replaced by C_{non} has become a conventional software verification problem; that is, the proof of the verification conditions implied by the assertions.

It is desirable to perform the verification condition generation (VCG) for this process alone, without also having to analyze the other processes and the main program (Figure 3-4). To do this, define **pause** by Figure 4-7 during VCG. For example, if we use symbolic execution for VCG, we symbolically execute the code in Figure 4-7 for each **pause**, introducing a new, unique symbolic variable upon each reference to *RandomValue*.

```
for i := 1 to NumOfNets do begin
    if $i \notin \{w_i\}$ then assign RandomValue to i;
    NetChanged[i] := NewNet[i] <> Net[i];
    Net[i] := NewNet[i];
    Updated[i] := FALSE
    end
```

Figure 4-7: Semantics of **pause** for VCG

An inductive assertion analysis of the process in Figure 4-6, using the **pause** semantics in Figure 4-7, proves that the assertions are always satisfied. The assertion at *L1* in turn implies that this process satisfies Eq. (4.7). This entitles us to replace the references to *NetChanged* by references to *WasUpdated* in the repeat condition. The resulting process never introduces case splits.

This section has shown how assertions and inductive assertion verification can be applied to prove Eq. (4.7), and thereby permit the application of the non-selective trace.

4.2.6. Another Generalized Process

In this application, assertions may tend to be cumbersome and redundant. The loop assertion in Figure 4-6 repeats the entire computation performed by the loop. This section describes a sufficient test of Eq. (4.7) that will avoid the generation of these assertions for some processes.

Figure 4-8 shows the new generalized process of interest. Figure 4-9 shows the same process, modified to conform to the form of Figure 4-4, with assertions added. Period "." denotes the current state. The term "$X\langle . \rangle$" denotes

$(NewNet[\{w_i\}]\langle F_i(.)\rangle = Net[\{w_i\}]\langle . \rangle)$ **and**

$(Locs_i\langle F_i(.)\rangle = Locs_i\langle . \rangle)$,

which is taken from Eq. (4.7).

```
begin
while TRUE do begin
    S;
    repeat pause until $\vee_{j \in \{r_i\}} NetChanged[j]$
    end
end
```

Figure 4-8: Generalized process for testing without assertions

Eq. (4.7) is satisfied if the assertions hold. Figure 4-10 shows the three paths that must be analyzed to verify the assertions. In this figure, "*Cond*" denotes

$\vee_{j \in \{r_i\}} NetChanged[j]$.

The **assume** statement conjoins the indicated condition with *!Pc*. Consider a treatment of the three paths by symbolic simulation.

Path (a): Symbolically execute path (a). At the end of the path, we must prove

```
begin
assert $TRUE$;
$S$;
while $TRUE$ do begin
    repeat pause;
        $L1$: assert (not $\bigvee_{j \in \{r_i\}} NetChanged[j]\langle.\rangle$)
                    $\supset X\langle.\rangle$;
        until $\bigvee_{j \in \{r_i\}} NetChanged[j]$;
    $S$
    end
end
```

Figure 4-9: Generalized process with assertions

```
assume $TRUE$;
$S$;
pause;
assert (not $Cond\langle.\rangle) \supset X\langle.\rangle$;
```

(a)

```
assume (not $Cond\langle.\rangle) \supset X\langle.\rangle$;
assume $Cond\langle.\rangle$;
$S$;
pause;
assert (not $Cond\langle.\rangle) \supset X\langle.\rangle$;
```

(b)

```
assume (not $Cond\langle.\rangle) \supset X\langle.\rangle$;
assume not $Cond\langle.\rangle$;
pause;
assert (not $Cond\langle.\rangle) \supset X\langle.\rangle$;
```

(c)

Figure 4-10: Three paths for VCG

$$!Pc\langle.\rangle \supset ((\textbf{not } Cond\langle.\rangle) \supset X\langle.\rangle) \quad .$$

This is equivalent to

$$(!Pc\langle.\rangle \text{ and } \textbf{not } Cond\langle.\rangle) \supset X\langle.\rangle \quad .$$

Therefore, we conjoin (not $Cond\langle.\rangle$) to the path condition $!Pc$, and then wish to show that $X\langle.\rangle$ holds. Condition $X\langle.\rangle$ includes references to $F_i(.)$, which we obtain by symbolically executing S. The complete test for path (a) is then as shown

in Figure 4-11. This test can be performed without first generating the assertions in Figure 4-9, although of course the test itself effectively generates the assertion. The symbolic execution of S is straightforward, since we have required that S introduce no case splits.

```
assume $TRUE$;
$S$;
pause;
assume not $\bigvee_{j \in \{r_i\}} NetChanged[j]\langle.\rangle$;
$s0 := .$;        (* save current state in auxiliary var. $s0$ *)
$S$;
assert ($NewNet[\{w_i\}]\langle.\rangle = Net[\{w_i\}]\langle s0\rangle$) and
      ($Locs_i\langle.\rangle = Locs_i\langle s0\rangle$);
```

Figure 4-11: Symbolic execution for test of path (a)

Path (b): Path (b) is identical to path (a) except for the initial path condition (the precondition). The precondition for path (b) implies the precondition for path (a), so path (b) is proven by the analysis of path (a). No further analysis is necessary.

Path (c): Consider a conventional execution of this path. The initial assumptions imply $X\langle.\rangle$. Either the pause will change one or more of $Net[\{r_i\}]$, or it will not. If it does, then $Cond$ is true following the pause, which implies that the final assertion is true. If the pause does not modify any of $Net[\{r_i\}]$, then X is true following the pause. This is because X was true before the pause, and the pause did not modify any of the inputs to process i. Then X true implies that the final assertion is again true.

No further analysis of path (c) is required; the definition of $Cond$ guarantees that path (c) is always correct.

Then path (c) is always correct, while the correctness of path (a) implies the correctness of path (b). Hence only path (a) need be tested. The symbolic execution in Figure 4-11 tests path (a) without requiring prior generation of the assertions in Figure 4-9. For example, this test will verify that non-selective trace may be used in Figure 4-1, without requiring the specification of the assertions shown in Figure 4-6.

In conclusion, given a process definition of the type in Figure 4-8, where S *never introduces* case splits, we may apply the symbolic execution in Figure 4-11. If the final assertion is proven, then we may use non-selective trace *instead of selective* trace, substituting *WasUpdated* for *NetChanged* in

$\bigvee_{j\in\{r_i\}} NetChanged[j]$.

The resulting process definition is equivalent to the old, and it never introduces case splits.

4.3. Non-selective Trace Summary

The use of selective *trace in event-driven* simulations often introduces a large number of case splits in a symbolic simulation. Sometimes, non-selective trace may be substituted for selective trace without *altering the function* performed by the process. The non-selective trace does not introduce case splits.

Non-selective trace may be substituted for selective trace in the process type typically used for combinational elements. The selective trace is used not to alter the functionality of the process, but rather to improve its efficiency in event-driven simulations. An analysis shows that the weaker non-selective trace may be used instead.

This substitution is also possible in the following more complex process code, where condition C refers to *NetChanged* (selective trace):

```
while TRUE do begin
    repeat pause; L1: until C;
    S
    end;
```

If C and S satisfy certain restrictions including Eq. (4.7),

$\forall$ valid start states $s|\ Lc_i\langle s\rangle = L1$:

$$(\text{not } C\langle s\rangle) \supset ((NewNet[\{w_i\}]\langle F_i(s)\rangle = Net[\{w_i\}]\langle s\rangle) \text{ and } (Locs_i\langle F_i(s)\rangle = Locs_i\langle s\rangle)) \quad . \tag{4.7}$$

then *WasUpdated* may be substituted for *NetChanged* in C. In this equation,

- Lc_i is the program counter for this process,
- $Locs_i$ denotes the local variables *for this process*,
- $\{w_i\}$ is the set of numbers of nets that can be written by this process, and
- F_i is a function describing the action of statement S.

The new process which accesses *WasUpdated* never introduces case splits. An inductive assertion analysis may be helpful in properly defining the set of "valid start states" so that Eq. (4.7) can be proven.

Alternately, a symbolic execution test is possible for processes of the following form:

```
while TRUE do begin
    S;
    repeat pause until $\bigvee_{j\in\{r_i\}} NetChanged[j]$
    end
```

Here, $\{r_i\}$ is the set of numbers of nets which can be read by this process. If the process satisfies the test, then we can substitute *WasUpdated* for *NetChanged* in the repeat condition. This test has the advantage of not requiring the use of assertions.

Our discussion did not conjecture or prove whether the test is necessary as well as sufficient.

Sometimes, a condition stronger than the simple non-selective condition is preferable. For example, in circuits with feedback, a stronger condition might prevent endless symbolic execution during periods of no actual activity in the circuit. This chapter gave one such stronger condition, which involves more computation than the simpler non-selective condition, but never introduces case splits.

Previous work did not apply symbolic execution to event-driven simulation. In particular, it did not investigate the substitution of non-selective trace for selective trace to reduce case splitting.

Chapter 5
Case Merging

In the previous chapter, we eliminated case splitting by replacing selective trace by non-selective trace. This is appropriate only in limited cases. In most cases, the process definition arising from this substitution is not equivalent to the original.

This chapter describes case merging, a more general technique for reducing case splitting. A case splitting simulation generates a new simulation for each alternative path at a conditional branch. A case merging simulation, by contrast, combines the effects of all possible alternatives into a single simulation.

The approach is to generate a new process equivalent to the original. The new process executes each basic block [AU77] from the original, subject to a path condition for each block. The new process includes the effects of conditional branches in the computation of conditional values for variables. The transformed process will never introduce case splits.

Cheatham et al. first suggested the use of conditional expressions to combine the results of several paths for a similar static program analysis [CH79]. This chapter applies this idea to the symbolic execution of event-driven simulations with the aid of additional analyses of a process's flow graph.

Section 5.1 shows case merging for a couple examples. Section 5.2 describes the application of case merging in general for processes in which every loop is broken by a **pause**. Section 5.3 presents the more complex analysis required if a process includes loops not broken by **pause**. Section 5.4 discusses a few optimizations which help to reduce expression complexity in symbolic execution.

The elimination of case splits does not occur without cost. Typically, the symbolic expressions which are generated by case merging are far more complex than those generated by case splitting. The next chapter will discuss this problem.

5.1. Case Merging Examples

5.1.1. Case Merging with Single Entry Point

Figure 5-1 shows a process that models an edge-triggered D flip-flop. In a conventional simulation, the **pause** loop waits until *Net*[*Clk*] is true immediately after a change in *Net*[*Clk*]. Hence, the **pause** waits for a positive transition in *Net*[*Clk*]. When this edge is detected, the flip-flop copies the *D* input to net *Q* and then waits for the next positive *Clk* edge.

```
process 102; (* Dff *)
const D = 210;     (* indices to Net *)
      Clk = 211;
      Q = 212;     (* index to NewNet *)
begin
while TRUE do begin
    repeat pause until NetChanged[Clk] and Net[Clk];
    assign Net[D] to Q
    end
end; (* 102 - Dff *)
```

Figure 5-1: Edge-triggered D flip-flop model

This process does not satisfy the restrictions from the previous chapter. If the statement

assign *Net*[*D*] **to** *Q*

were executed when the **repeat** condition was false, it could alter *NewNet*[*Q*], violating Eq. (4.7). Therefore, we cannot simply substitute *WasUpdated* for *NetChanged* in the **pause** loop to eliminate case splits. Instead, we will transform the process to eliminate case splits by another method.

Figure 5-2 shows a graph representation of the process. Vertices represent basic blocks as defined by Aho and Ullman [AU77], with the added provision that a **pause** itself constitutes a separate basic block that appears twice in the graph, once as an entry point and once as an exit point. Conditions on edges out of a vertex denote the conditions under which those edges are followed. This graph is the flow graph defined by Aho and Ullman for dataflow analysis [AU77].

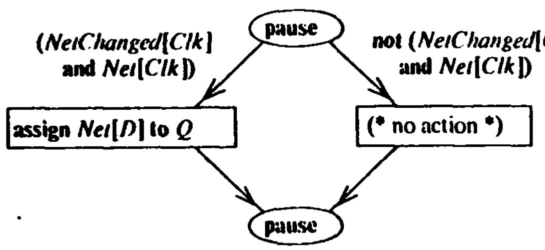

pause
(NetChanged[Clk] and Net[Clk])
not (NetChanged[Clk] and Net[Clk])
assign Net[D] to Q
(* no action *)
pause

Figure 5-2: Flow graph for D flip-flop process

The action of the model in Figure 5-2 on *Net*[*Q*] can be characterized by the statement

assign (*Aux1* **and** *Net*[*D*]) **or** (**not** *Aux1* **and** *Net*[*Q*]) **to** *Q* ,

where *Aux1* is *NetChanged*[*Clk*] **and** *Net*[*Clk*]. The method for deriving this result from the flow graph is covered in the more general discussion later. This example is simple enough that the result is apparent by inspection. Figure 5-3 then shows the transformed version of the D flip-flop process. In this transformed version, the **pause** loop condition could be any condition C_x such that

$$(NetChanged[Clk] \text{ and } Net[Clk]) \supset C_x \quad .$$

Figure 5-3 uses

$$C_x = WasUpdated[Clk] \quad .$$

This process is equivalent to that in Figure 5-1, and it never causes a case split. The results of two parallel paths thus are combined into a single simulation.

```
process 102; (* Dff *)
const D = 210;      (* indices to Net *)
      Clk = 211;
      Q = 212;      (* index to NewNet *)
var   Aux1: Boolean;
begin
while TRUE do begin
    repeat pause until WasUpdated[Clk];
    Aux1 := NetChanged[Clk] and Net[Clk];
    assign (Aux1 and Net[D]) or (not Aux1 and Net[Q]) to Q
    end
begin; (* 102 - Dff *)
```

Figure 5-3: Transformed version of D flip-flop

The claim that the process in Figure 5-3 never introduces case splits depends on the value of *WasUpdated*[*Clk*] not being symbolic. However, case merging itself appears to put this condition in jeopardy: One path might make an assignment to net *j* while a parallel path makes no such assignment. If these two paths are combined into a single simulation by case merging, then the value of *WasUpdated*[*j*] should be symbolic, denoting the condition under which a value was assigned.

For example, the **assign** in Figure 5-3 sets *Updated*[*Q*] (which becomes *WasUpdated*[*Q*]) to *TRUE*, when it would be more accurate to set *Updated*[*Q*] to *Aux1*. This is because, in Figure 5-2, the process makes an assignment to net *Q* if and only if *Aux1* is true.

The solution to this problem is to recall that the simulation results are by definition independent of *WasUpdated*. Therefore we can modify the management of *WasUpdated*. The only requirement is that we must have

$$NetChanged[j] \supset WasUpdated[j] \quad .$$

This requirement is due to our use of *WasUpdated* above and in the previous chapter to reduce case splitting. Our strategy will be to set *Updated*[*j*] to *TRUE* if *any* path included in a case merging simulation performs an **assign** to net *j*.

Finally, note that the first time the process in Figure 5-1 is called, its execution proceeds from the beginning of the process, not from the **pause**. Our discussion above did not take this into account. Fortunately, the result is satisfactory, because on this first entry, we know (from Figure 3-4) that *NetChanged*[*Clk*] is false. Therefore the action taken by the process starting from its beginning is the same as if it started from the **pause**. In both cases, the process goes to the **pause** with no modifications to *D*. In the next example, we will not rely on such a coincidence.

5.1.2. Case Merging with Multiple Entry Points

The example in Figure 5-4 copies *Net*[*A*] and *Net*[*B*] to the output net on rising and falling edges of net *X*, respectively. This is similar to the double-edge-triggered flip-flop suggested by Unger [Un81]. This description will demonstrate a complication in the elimination of case splitting.

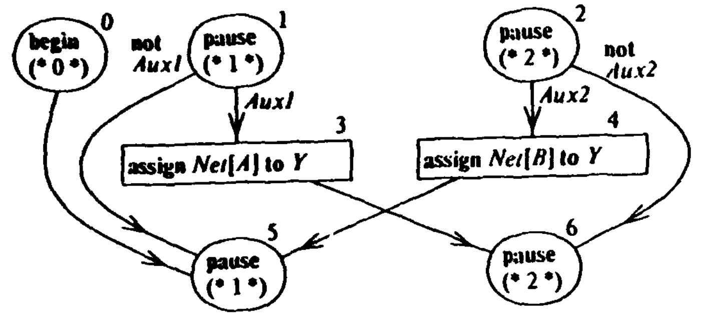
```
process 103; (* OddReg *)
const A = 220;        B = 221;
      X = 222;        Y = 223;
begin                                                          (* 0 *)
while TRUE do begin
    repeat pause until NetChanged[X] and Net[X];               (* 1 *)
    assign Net[A] to Y;
    repeat pause until NetChanged[X] and not Net[X];           (* 2 *)
    assign Net[B] to Y
    end
end; (* 103 - OddReg *)
```

Figure 5-4: Process with multiple entry points

The numbered comments in Figure 5-4 mark places at which the process may begin execution in a **cobegin**, namely its beginning and after a **pause**. In Figure 5-1, the process has only one such entry point, namely the **pause**. (Recall that in the D flip-flop, starting from the beginning was equivalent to starting from the **pause**, so there was effectively only one entry point.)

Figure 5-5 shows the flow graph for this process. By convention, the beginning of a process (the first entry point) appears as a single basic block with no predecessors. The presence of multiple entry and exit points complicate the analysis. We will introduce an extra array *LPc* containing one Boolean element for each vertex (basic block) in the graph. The number labeling each vertex in

Figure 5-5 denotes the corresponding element of *LPc*. Note that each pause gets two such indices; one for its function as an entry and one for its function as an exit. Figure 5-6 then shows the transformed process, described below.



Aux1 = *NetChanged*[*X*] and *Net*[*X*]
Aux2 = *NetChanged*[*X*] and not *Net*[*X*]

Figure 5-5: Flow graph for *OddReg* process

LPc[*j*] becomes true if and only if basic block *j* is to be run during this execution of the process. *LPc*[*j*] may be computed from the values of *LPc* for the predecessors of block *j*, taking into account any conditions associated with the edges. A topological sort on the graph yields an appropriate order for the processing of the blocks, since the graph is acyclic. The order in which the vertices are numbered happens to be a valid topological sort, and so is the order we use. The value assigned to net *Y* is then computed using the values of *LPc* computed for the various basic blocks.

```
process 103; (* OddReg *)
const A = 220;          B = 221;
      X = 222;          Y = 223;
var   Aux1, Aux2: Boolean;
      LPc: array [0..6] of Boolean;
begin
LPc[0] := TRUE;
LPc[1] := FALSE;
LPc[2] := FALSE;
while TRUE do begin
    Aux1 := NetChanged[X] and Net[X];
    Aux2 := NetChanged[X] and not Net[X];
    LPc[3] := LPc[1] and Aux1;
    LPc[4] := LPc[2] and Aux2;
    LPc[5] := LPc[0] or LPc[1] and not Aux1 or LPc[4];
    LPc[6] := LPc[3] or LPc[2] and not Aux2;
    assign (LPc[3] and Net[A]) or (LPc[4] and Net[B]) or
          ((LPc[0] or LPc[1] and not Aux1 or LPc[2] and not Aux2) and
           Net[Y])
                to Y;
    LPc[0] := FALSE;
    LPc[1] := LPc[5];          LPc[2] := LPc[6];
    repeat pause until WasUpdated[X]
    end
end; (* 103 - OddReg *)
```

Figure 5-6: Modified process with multiple entry points

LPc[5..6] become the initial values of *LPc*[1..2] for the next execution of the process. The pause loop condition may be any function C_x such that

$(Aux1 \text{ or } Aux2) \supset C_x$.

We choose C_x equal to *WasUpdated*[*X*].

Now, the process in Figure 5-6 can be executed symbolically without case splitting. This is true since the only conditional branch expression in the process is *WasUpdated*[*X*], and by convention *WasUpdated* will never have symbolic values in a symbolic simulation. (Recall discussion near end of previous section.)

Note that in a symbolic simulation, *LPc* may have symbolic (not constant) values. This implies that in a case merging symbolic simulation, we cannot uniquely identify the path executed by a process during a given **cobegin** call. By the same token, we can assign symbolic initial values to *LPc*, yielding a more general symbolic execution.

A later section will discuss the derivation of the assignments to *LPc* and net *Y* in Figure 5-6.

5.1.3. Summary of Examples

The above case merging examples cannot eliminate case splitting simply by substituting non-selective trace for selective trace. Rather, each example combines the results of all possible paths into a single simulation. The second example shows that in general, the simulator may have to combine paths with different starting and stopping points, since the path followed by the process is not uniquely identified.

The following sections will describe the application of this technique to achieve symbolic simulation without case splitting for a large class of process types. The major area of concern is how to characterize the behavior of the process given the flow graph.

5.2. Analysis of Acyclic Flow Graphs

The generation of a flow graph like those in Figures 5-2 and 5-5 is straightforward. The graph will be acyclic if every loop in the process is broken by a **pause**. If the graph is acyclic, we can construct a new process according to the following procedure, illustrated with the *OddReg* example. This construction yields

an equivalent process of a form similar to that in Figure 5-6. This resulting process can be executed symbolically without case splitting.

The strategy will be to construct a process that executes every basic block from the original process once. Each basic block will use the results from any preceding basic blocks. Since every basic block is executed, there will be no need for conditional branches to decide which blocks to execute. The removal of the conditional branches eliminates case splitting. The only remaining conditional branch will be in the **pause** loop.

5.2.1. Generality of the Construction

This construction is applicable to any process satisfying two restrictions:

- Every loop is broken by a **pause**.
- During symbolic execution, every subscript evaluates to a constant.

If the first restriction is not satisfied, then a more general analysis is required. This will be considered later in this chapter. The second restriction allows us to ignore the problems of case splitting due to subscript evaluation. If it is not satisfied, we can rewrite the code to satisfy the restriction, or we can use more complex symbolic expressions that permit non-constant subscripts.

For example, in the expression

Reg[*RegSpec*] ,

the subscript *RegSpec* must evaluate to a constant. However, it is common for *RegSpec* to be part of a symbolically specified machine instruction that does not evaluate to a constant.

If the subscript range is small, then the subscripted reference can be replaced by a case statement which extracts the correct value. If there were only two registers, we could write the following:

```
case RegSpec of
    0:  RSubR := Reg[0];
    1:  RSubR := Reg[1]
    end;
```

Subsequent references to *Reg*[*RegSpec*] would then be replaced by "*RSubR*".

If the subscript range is not small, then the subscript must evaluate to a constant, a serious limitation in some cases. We can lift this restriction by using more powerful symbolic representations of data structures, such as those used by Luckham and Suzuki for Pascal [LS79] (originally proposed by McCarthy for a different application [Mc62]). This in turn would require more powerful symbolic expression manipulation.

5.2.2. The Construction

This section describes the generation of code equivalent to the process represented by the flow graph, but with no conditional branches except in the pause loop. The result of this construction for *OddReg* appears in Figure 5-7.

Every vertex (basic block) in the flow graph will have its own copy of every variable that the process can modify. In *OddReg*, the only such variables are *NewNet*[*Y*] and *Updated*[*Y*]. According to the convention noted earlier, we will not compute a symbolic value for *Updated*[*Y*] in symbolic simulations (it will be set true), so ignore it here. Let *NewY*[0..6] be the array of copies of *NewNet*[*Y*] for the several basic blocks.

In addition to the multiple copies of modified variables, we also have Boolean array *LPc*, which holds a "local path condition" for each basic block. Initially, the element of *LPc* corresponding to the beginning of the process (the first entry point) is *TRUE*, while the elements for the other entry points are *FALSE*.

The remaining code to be generated becomes the body of a **while** *TRUE* **do** loop.

Consider the blocks in any order that is a valid topological sort. The reason this order is important is that basic block j needs the final values of variables from its immediate predecessors to initialize its own variables. Ordering the construction by a topological sort ensures that these final values will be available. The topological sort is of complexity $O(V, E)$, where V is the number of vertices (basic blocks) and E is the number of edges (branches between basic blocks) [AH74].

For each basic block j in order, generate the following three sections of code:

1. Code to compute $LPc[j]$.

If block j is an entry point (no predecessors), then $LPc[j]$ is already set and need not be computed. Otherwise, let $\{b_j\}$ be the numbers of the predecessors of j. Let $C(k,j)$ be the condition on the edge from block k to j. If no such condition is given for edge $\langle k,j \rangle$, then $C(k,j)$ is *TRUE*. Then $LPc[j]$ is given by the following:

$$LPc[j] := \mathbf{or}_{k \in \{b_j\}}(LPc[k] \ \mathbf{and} \ C(k,j))$$

For example, $LPc[6]$ in Figure 5-5 is found by

$$LPc[6] := LPc[3] \ \mathbf{or} \ LPc[2] \ \mathbf{and\ not} \ Aux2$$

If $C(k,j)$ reads any variables modified by this process, it reads the copies local to block k.

2. Code to initialize the local copies of variables for block j.

If block j is an entry point, then the local copies of variables are copied from the actual variables. For example, the initialization for block 1 in Figure 5-5 is simply

$NewY[1] := NewNet[Y]$.

(To observe the restriction against reading *NewNet*, substitute *Net* for *NewNet* here; the two are equal at this point.)

Otherwise, block j must initialize its local variables from the variables in its predecessor blocks. If block j has more than one predecessor, then the values computed will be conditional, denoting several possible alternatives computed in different paths. For example, the value computed for $NewY[6]$ in Figure 5-5 is

$NewY[6] := LPc[3]$ **and** $NewY[3]$ **or** $LPc[2]$ **and not** $Aux2$ **and** $NewY[2]$.

Note the similarity between this assignment and the computation of $LPc[6]$.

3. Code to execute block j.

The code from block j can be used directly, with two modifications: (i) Any references to variables that can be modified by this process are replaced by references to the copies local to this basic block. (ii) The **assign** statement is a special case of the first modification. This reference to *NewNet* and *Updated* is replaced by a simple assignment to the local copy of *NewNet*. For example, the code for block 4 in Figure 5-5 becomes the following:

$NewY[4] := Net[B]$

We must remember to set the appropriate elements of *Updated* to *TRUE* later.

Following the above generation of code for all the basic blocks, generate code to merge the results from the several exit blocks. This merging is similar to the initialization of local variables in a block with predecessors, except now the variables being written are the actual variables, not local copies. Various **assign** statements will update *NewNet* and set appropriate elements of *Updated* to *TRUE*.

Next, generate code to copy the values of *LPc* for the exit blocks back to the corresponding elements for the entry blocks. The element of *LPc* for the initial entry point is set false here.

Finally, generate a pause loop that terminates this call to the process. The pause loop condition may be any condition that is guaranteed to be satisfied whenever any of the pause loop conditions from the original process is satisfied. The condition also should never cause a case split in a symbolic simulation. A condition satisfying both of these requirements can always be found; in the worst case, the condition can be *TRUE*.

The above construction carried out for *OddReg* yields the process in Figure 5-7.

```
process 103; (* OddReg *)
const A = 220;          B = 221;
      X = 222;          Y = 223;
var   LPc, NewY: array [0..6] of Boolean;
LPc[0] := TRUE;           LPc[1] := FALSE;
LPc[2] := FALSE;
while TRUE do begin
     NewY[0] := Net[Y];
     NewY[1] := Net[Y];
     NewY[2] := Net[Y];
     LPc[3] := LPc[1] and NetChanged[X] and Net[X];
     NewY[3] := NewY[1];
     NewY[3] := Net[A];
     LPc[4] := LPc[2] and NetChanged[X] and not Net[X];
     NewY[4] := NewY[2];
     NewY[4] := Net[B];
     LPc[5] := LPc[0] or LPc[1] and not (NetChanged[X] and Net[X])
                    or LPc[4];
     NewY[5] := LPc[0] and NewY[0] or
                LPc[1] and not (NetChanged[X] and Net[X]) and NewY[1] or
                LPc[4] and NewY[4];
     LPc[6] := LPc[3] or LPc[2] and not (NetChanged[X] and not Net[X]);
     NewY[6] := LPc[3] and NewY[3] or
                LPc[2] and not (NetChanged[X] and not Net[X]) and NewY[2];
     assign LPc[5] and NewY[5] or LPc[6] and NewY[6] to Y;
     LPc[0] := FALSE;
     LPc[1] := LPc[5];          LPc[2] := LPc[6];
     repeat pause until WasUpdated[X]
     end
end; (* 103 - OddReg *)
```

Figure 5-7: Algorithmically generated process for case merging

5.2.3. Use of Constructed Process

A process resulting from the above construction, for example the one in Figure 5-7, can be used as the final form of the process. It can be executed symbolically without case splitting. In the symbolic execution, array *LPc* will have symbolic values which can be considered as being path conditions for their associated basic blocks.

Alternately, we may symbolically execute this process independently and use the symbolic results to define a new version of the process. In brief, this is done by (i) initializing all program variable values (including *LPc*) to distinct symbolic variables, (ii) symbolically executing the body of the loop once, and (iii) generating *assignment statements* for the several variables from their resulting symbolic values.

The latter option has the advantages of perhaps eliminating many intermediate computations and allowing a one-time optimization of the symbolic expressions generated by this process. The process given in Figure 5-6 represents a mixture of these two options. The computations for array *LPc* in Figure 5-6 are taken directly from Figure 5-7; the only change is the use of *auxiliary variables* *Aux1* and *Aux2*. But the computation of *NewNet*[*Y*] in Figure 5-6 is a substantial abbreviation of the computation in Figure 5-7.

5.2.4. Application of Construction to Routines

The above construction can also be applied to function and procedure ("routine") definitions as well as to process definitions. While a process has entry and exit points at each pause plus an *entry point at its beginning*, a routine will have just one entry point, at its beginning, and one exit point, at its end. The construction then is the same as for processes, except that the entry element of the routine's *LPc* array is set equal to the value of *LPc* for the calling basic block.

As with processes, the result of the construction can itself be the final version, or it can be symbolically executed to generate a shorter final version. In the latter case, one must take care to account for possible anomalous parameter bindings, such as the binding of one variable to two distinct call-by-reference formals in the same call.

5.3. Analysis of Flow Graphs with Loops

Figure 5-8 shows an outline of the process generated by the construction of Section 5.2.2. The generated process executes each basic block from the original process exactly once. This construction is not adequate for processes with loops not broken by **pause**. Basic blocks in such loops may be executed more than once in one call to the process. We need a mechanism that will allow the execution of basic blocks more than once.

```
init elements of $LPc$ for entry basic blocks;
while $TRUE$ do begin
    for each basic block $j$, taking the blocks in topological order,
        do begin
            set $LPc[j]$;
            init $Vars[j]$ from $Vars[\{b_j\}]$;
            execute block $j$
            end;
    update vars modified by process, including assigns;
    copy $LPc$ elements for entry blocks from elements for exit blocks;
    repeat pause until $Condition$
    end;
```

Figure 5-8: Case merging outline for process without loops

One possibility is to analyze the loop to determine a closed-form representation of its computation. An inductive assertion analysis or the solution of recurrence relations [CH79] are possible approaches. This is an open research problem which we will not consider here.

A second possibility is to modify the outline of Figure 5-8 to handle loops. The remainder of the section discusses this approach.

5.3.1. New Outline for Case Merging with Loops

In Section 5.2.2, the choice of the next basic block is statically determined by the structure of the generated process. Figure 5-9 shows an equivalent outline that uses a dynamic mechanism instead. This figure marks lines that differ from the outline of Figure 5-8. *BBSet* is a set of integers. We have $j \in BBSet$ when one of j's predecessor blocks has been executed. This indicates that block j should be executed. Set *BBSet* is managed as a priority queue, where the priority of each block is determined by its position in a topological sort. Recall the motivation for topological sort from Section 5.2.2.

```
init elements of $LPc$ for entry basic blocks;
while $TRUE$ do begin                                                    5-8 |
    init $BBSet$ to include entry basic block numbers;                   5-8 |
    while $BBSet$ includes numbers of blocks that are not exit blocks
        do begin                                                         5-8 |
            choose and remove non-exit block $j$ from $BBSet$;
            set $LPc[j]$;
            init $Vars[j]$ from $Vars[\{b_j\}]$;
            execute block $j$;                                           5-8 |
            add to $BBSet$ the numbers of successor blocks
            end;
    update vars modified by process, including assigns;
    copy $LPc$ elements for entry blocks from elements for exit blocks;
    repeat pause until $Condition$
    end;
```

Figure 5-9: Equivalent case merging version using basic block set

Set *BBSet* initially contains only the numbers of the entry blocks. When a block is selected, the simulator removes its number from *BBSet* and adds the numbers of its successors. Execution continues until *BBSet* contains only numbers of exit blocks.

The outline of Figure 5-9 still cannot handle processes with loops. Figure 5-10 presents a modified outline that corrects the problems noted below.

```
 1 init elements of LPc for entry basic blocks;
 2 while TRUE do begin
 3     init non-entry LPc elements to FALSE;                        5-9 |
 4     init Vars elements to FALSE (empty);                         5-9 |
 5     init BBSet to include entry basic block numbers;
 6     while BBSet includes numbers of blocks that are not exit blocks
 7         do begin
 8             choose and remove non-exit block j from BBSet;
 9             set UseLPc;                                          5-9 |
10             if UseLPc is satisfiable then begin                  5-9 |
11                 init UseVars from Vars[{b_j}];                   5-9 |
12                 adjust Vars[{b_j}];                              5-9 |
13                 ∀k∈b_j do LPc[k] := LPc[k] and not C(k,j);       5-9 |
14                 execute block j;
15                 add to BBSet the numbers of successor blocks;
16                 Vars[j] := Vars[j] or UseVars and UseLPc;        5-9 |
17                 LPc[j] := LPc[j] or UseLPc                       5-9 |
18             end                                                  5-9 |
19         end;
20     update vars modified by process, including assigns;
21     copy LPc elements for entry blocks from LPc for exit blocks;
22     repeat pause until Condition
23     end;
```

Figure 5-10: Case merging outline for process with loops

First, the outlines of Figures 5-8 and 5-9 include no provision for omitting the execution of a basic block j which has $UseLPc \equiv FALSE$. (The use of $UseLPc$ instead of $LPc[j]$ is explained later.) This is merely inefficient when the original process is loop-free. But if the process has loops, this problem prevents the termination of the loop. Line 10 in Figure 5-10 makes the necessary check. For conventional execution, $UseLPc$ is always $TRUE$ or $FALSE$, so the following test would suffice:

if *UseLPc* **then begin** . . .

However, we want to be able to symbolically execute this process without case splitting. This explains the use of the more complex test:

if *UseLPc* is satisfiable **then begin** . . .

The end of this section makes further observations on the use of this test.

Second, the presence of loops implies that a basic block j can be executed before one of its predecessors b_j has been executed. In this case, $LPc[b_j]$ should be $FALSE$. Line 3 in Figure 5-10 initializes LPc to ensure this is true.

Third, the presence of loops implies that a basic block j may be executed more than once. Suppose block k is a predecessor of block j. If condition

$LPc[k]$ **and** $C(k,j)$

is included in the first execution of block j, then it should not also be included in subsequent executions of block j. In Figure 5-10, line 13 makes the necessary adjustment to $LPc[k]$ to prevent this problem.

Finally, the second execution of a basic block should not destroy the results of its first execution. This implies that the basic block requires two local copies of variables that can be modified. In Figure 5-10, *UseVars* is the copy of variables for the current computation, while *Vars* accumulates and saves the results from previous computations for later use. Similarly, each basic block has two copies of the path condition, *UseLPc* and *LPc*. But since only one basic block is executed at a time, then a separate copy of *UseVars* and *UseLPc* is not required for each basic block. *UseVars* and *UseLPc* need not be arrays.

Line 4 initializes the elements of *Vars* to reflect the fact that no results have been accumulated yet. Lines 9 and 11 set *UseLPc* and *UseVars*, not *LPc*[*j*] and *Vars*[*j*], for current use. The adjustment of *Vars*[$\{b_j\}$] in line 12 is similar to the adjustment of *LPc*[$\{b_j\}$] in line 13, discussed above. Lines 16 and 17 merge the results *UseVars* and *UseLPc* into *Vars*[*j*] and *LPc*[*j*].

This completes the modification of the case merging outline to support processes with loops. The only remaining detail is the selection of the next basic block from *BBSe*[illegible] execution, covered in the next section. The remainder of this section discusses further the satisfiability test for *UseLPc* in line 10.

Consider a symbolic execution of this process in which *UseLPc* is satisfiable but not identically true. Since *UseLPc* is satisfiable, block *j* will execute. Now we are not introducing a case split even though the condition is not identically true. This implies that this symbolic execution represents some conventional executions which would not execute block *j* at this point. The symbolic execution must predict no changes due to the block for such cases.

Block *j* modifies *UseLPc* and *UseVars*. However, each block initializes these variables itself before using them, so any modifications to *UseLPc* and *UseVars* by block *j* are immaterial. Block *j* modifies *LPc*[*j*] and *Vars*[*j*], but it computes conditional values, so that (not *UseLPc*) implies that *LPc*[*j*] and *Vars*[*j*] do not change. Similarly, the conditional adjustment to *Vars*[$\{b_j\}$] and *LPc*[$\{b_j\}$] in lines 12 and 13 ensure that these variables are not changed when *UseLPc* is false. For *LPc*[*k*], $k \in \{b_j\}$, we have

$$\begin{aligned}(\text{not } UseLPc) &\Rightarrow \text{not } (LPc[k] \text{ and } C(k,j))\\ &\Rightarrow (LPc[k] \supset \text{not } C(k,j))\\ &\Rightarrow ((LPc[k] \text{ and not } C(k,j)) \equiv LPc[k])\end{aligned}$$

The argument for *Vars*[$\{b_j\}$] is similar.

The other issue concerning the satisfiability test on *UseLPc* is its effect on the generality of this approach to case merging. For a loop to terminate during symbolic execution, *LPc*[*j*] for every block *j* in the loop must eventually become unsatisfiable. This implies that the loop must unfold during symbolic execution; that is, there must be a known finite bound on the number of passes through the loop. A similar restriction applies to loops implemented by recursive routine calls; otherwise, a routine may call itself endlessly, never terminating. These restrictions limit the generality of processes to which case merging as described in this section can be applied.

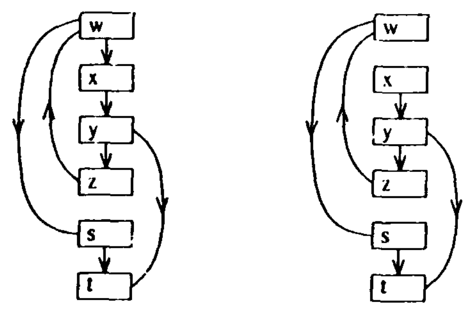
Note that this technique for symbolically executing loops is likely to generate complex symbolic expressions, especially if the loop condition does not resolve to *TRUE* or *FALSE* on each pass through the loop.

5.3.2. Selection of Basic Blocks for Execution

For loop-free processes, a topological sort yields appropriate priorities for use in Figure 5-10. However, in processes with loops, a topological sort cannot be found. In this case, it is desirable to find a priority assignment which will tend to minimize the numbers of times basic blocks are executed. Given a flow graph for a process, a possible heuristic is to remove edges until the graph is acyclic. Then carry out a topological sort on the graph to determine block priorities.

The guiding principle for the removal of edges is as follows: In the topological sort, all vertices (basic blocks) in the body of the loop should precede vertices for code following the loop. This ensures that when the process is in a loop, it will completely exit the loop before proceeding from the loop. The choice of edges removed from the graph should ensure that any topological sort will satisfy this property.

For example, in Figure 5-11a, the simulator should select block s from *BBSet* only if none of blocks w, x, y, or z remain in *BBSet*. Figure 5-11b shows the graph from Figure 5-11a with edge $\langle w, x\rangle$ removed. All of vertices w, x, y, and z will precede vertex s in any topological sort of a graph containing this subgraph.



(a) Graph with loop (b) Loop broken

Figure 5-11: Graph with loop not broken by dismissal point

The following algorithm describes the assignment of vertex (block) priorities in graphs containing loops. It selectively removes edges until the graph is loop-free, at which point a topological sort gives the vertex priorities. This algorithm exploits

the fact that Tarjan's algorithm to find strongly connected components [AH74] directly yields a topological sort when applied to a loop-free graph.

1. Delete all self-loops $\langle v, v\rangle$ in the flow graph for the process. Call the resulting graph G_{pr}.

2. Find the strongly connected components in the G_{pr}. Assign each component a unique number. Let COMP(v) denote the number of the component containing vertex v. The component numbers should be ordered so that COMP(v) $\geq$ COMP(w) *iff* $v \rightarrow^* w$. Tarjan's algorithm for finding strongly connected components is $O(V, E)$ (linear on the number of vertices and edges) [AH74]. It also easily yields component numbers satisfying the above ordering.

3. Any component which contains more than a single vertex has a loop. For each such component i, do steps 4 and 5.

4. Let m be the largest integer such that $i > m$ and there exists an edge $\langle v, w\rangle$ in G_{pr} with v in component i and w in component m. Component m is a closest successor of component i, in the sense that if $v \rightarrow^* x \rightarrow^* w$, then vertex x is either in component i or in component m.

If no such integer m exists, then a simulator cannot exit component i, so that execution of this process would never terminate if it entered this loop. Therefore, we assume that such an integer can be found.

5. Choose a vertex s in component i such that there exists an edge $\langle s, t\rangle$ with COMP(t) = m. Remove from the graph all edges $\langle s, u\rangle$ for which COMP(u) = i.

6. Repeat steps 2 through 5 until step 2 finds no strongly connected component with more than a single vertex. *When this occurs,* the component numbering generated by step 2 is the priority assignment to be used for G_{pr}: the higher the value of COMP(v), the higher the priority of vertex (basic block) v.

We can apply this procedure to the graph in Figure 5-11a. This graph has no self-loops. After step 2, component number 1 has vertex t, component 2 has vertex s, and component 3 has vertices w, x, y, and z. Only component 3 has more than one vertex. For this component, the closest successor is component 2. Edge $\langle w, s\rangle$ is incident out of component 3 and into component 2. Edge $\langle w, x\rangle$ is incident out of w inside component 3, so we remove that edge. The first execution of steps 2 through 5 then yields the graph in Figure 5-11b.

The next execution of step 2 does not find any component with more than a single vertex. The vertex ordering generated by step 2 is (highest priority to lowest) w, x, y, z, s, t. This ordering guarantees that the simulator will not select label s or t unless none of w, x, y, and z are in the queue.

This procedure can be summarized as shown in Figure 5-12. Each pass through the **repeat** loop takes $O(V, E)$ time. In the worst case, $O(V)$ passes through the **repeat** are required. This is so because

- each execution of step 5 will cause vertex s to be a separate component in the next pass,
- therefore each execution of step 5 increases the number of components by at least one,
- *the maximum number of components is* $O(V)$, *and*

- an example can be constructed in which $O(V)$ passes are actually *required.*

```
begin (* priority assignment *)
remove self-loops ⟨v,v⟩ from graph, yielding graph $G_{pr}$;
repeat
    find strongly connected components of $G_{pr}$;
        numbering components so that
        component j is reachable from a different component i
        only if i > j;
    tag each vertex with its component number;
    for each component i with more than one vertex do begin
        find edge ⟨s, t⟩ with s in component i,
            t in component m, a closest successor of component i;
        remove all edges ⟨s, u⟩ for which u is in component i
        end
    until no component has more than one vertex;
current component numbering provides desired basic block priorities
end; (* priority assignment *)
```

Figure 5-12: *Priority assignment for graphs with loops*

The assignment of priorities to labels is then $O(V^2, E)$ in the worst case, although probably much faster if the control flow is structured and loops are not nested deeply. In structured process descriptions, fast (linear) algorithms might take advantage of the regular structure present in the original process definition, entirely bypassing the generation of the flow graph.

5.4. Optimizations

The new processes generated by a case merging construction will compute complex symbolic expressions. Therefore optimization of the constructed processes will be important. Whereas optimizations typically aim to reduce time or space requirements for code, optimizations of the case merging processes for symbolic simulation would aim to reduce the complexity of symbolic expressions, since the comparison of symbolic Boolean expressions is np-hard, for example.

Following is a brief description of some possible optimizations. The list is non-exhaustive. Algorithmic application of the optimizations is not discussed. Opportunities for these optimizations may be found by heuristics or dataflow analysis.

In Figure 5-13, the case merging construction would generate

$LPc[5] :=$ $LPc[4]$ **or** $LPc[2]$ **and not** $Aux2$ **or** $LPc[3]$.

But from the figure, it is clear that

$LPc[5] := LPc[1]$

is an equivalent and more desirable alternative. Similarly, if the basic blocks have local copies $QX[j]$ of variable X, but blocks 2, 3, and 4 do not modify X, then the assignment

$QX[5] :=$ $LPc[4]$ **and** $QX[4]$ **or** $LPc[2]$ **and not** $Aux2$ **and** $QX[2]$
or $LPc[3]$ **and** $QX[3]$

can be optimized to

$QX[5] := QX[1]$.

Subtler possibilities exist. If a variable T is written only in block 1 and read only in blocks 2, 3, 4, and/or 5, then local copies of T are not required at all.

In Figure 5-14, assume the blocks have local copies of variable Z. According to the case merging construction, block 3 computes the following conditional expressions:

$LPc[3] :=$ $LPc[1]$ **and** $Aux1$ **or** $LPc[2]$;
$QZ[3] :=$ $LPc[1]$ **and** $Aux1$ **and** $QZ[1]$ **or** $LPc[2]$ **and** $QZ[2]$;

Subsequently, block 5 will compute a conditional value for $QZ[5]$:

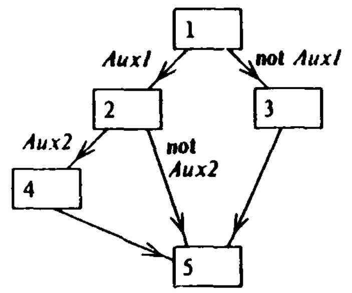


Figure 5-13: Example for optimizations

$QZ[5] :=$ $LPc[3]$ **and** $Aux2$ **and** $QZ[3]$ **or** $LPc[4]$ **and** $QZ[4]$;

Now, if block 3 did not modify Z, then we can substitute for $LPc[3]$ and $QZ[3]$, finding the following:

$QZ[5] :=$ ($LPc[1]$ **and** $Aux1$ **or** $LPc[2]$) **and** $Aux2$ **and**
($LPc[1]$ **and** $Aux1$ **and** $QZ[1]$ **or**
$LPc[2]$ **and** $QZ[2]$) **or**
$LPc[4]$ **and** $QZ[4]$;

From this we see that anding ($LPc[3]$ **and** $Aux2$) with $QZ[3]$ is a redundant operation in the computation of $QZ[5]$. ($Aux2$ is redundant because only one path goes from block 3 to block 5; compare to the computation of $QX[5]$ in Figure 5-13, with two paths from block 2 to block 5.) Another example of this appears in the computation of the value assigned to Y in Figure 5-7. The case merging construction for processes with loops aggravates this problem, since $Vars$ values are always conditional expressions (line 16 in Figure 5-10).

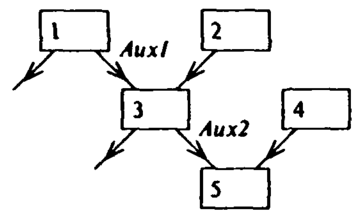


Figure 5-14: Second example for optimizations

5.5. Case Merging Summary

This chapter has described the symbolic execution of processes without case splitting in an event-driven simulation. The approach is to create a flow graph for the process of interest. In this graph, each path from an entry point to an exit point represents a possible execution path for the process during one cobegin statement. Then from the graph, generate a new process equivalent to the original. The discussion illustrated this approach with two examples, and then described it for the general case.

If the graph is loop-free, then the new process executes each basic block from the original process exactly once. The new process contains no conditional branches except for one in a **pause** loop that does not introduce a case split when symbolically executed. The new process can be symbolically executed without case splitting. It therefore combines all possible paths as represented by the several paths in the graph; hence the name "case merging."

If the graph is not loop-free, a different construction yields a new process

which may execute some basic blocks from the original process more than once. The new process contains a conditional branch in the processing of each block in addition to the one in the **pause** loop. The new branch conditions test for the *satisfiability* of an expression, and so *do not introduce* case splits in a symbolic execution. A symbolic execution of this new process is guaranteed to terminate only if all loops in the original process unfold during symbolic execution. This restricts the generality of this approach for processes with loops.

In the case of graphs with loops, a criterion is needed for choosing the order in which the basic blocks *from the original process* will be executed. A heuristic selectively removes edges from the flow graph until it is loop-free. A (static) priority is assigned to each basic block according to a topological sort on the graph. A priority queue manager then controls the selection of basic blocks for execution.

This chapter also described a few optimizations which may reduce the complexity of the symbolic expressions generated in a case merging simulation.

Cheatham, Holloway, and Townley used conditional expressions for case merging in another context [CH79]. They built a similar graph model and used conditional expressions to characterize the effects of multiple paths in a static program analysis. This chapter added the following enhancements to their work:

- treatment of processes with multiple entry points (essentially coroutines); use of topological sort to order the analysis of such routines,
- extraction of loop structure in Section 5.3.2,
- optimizations from Section 5.4.

Cheatham et al. did not include a comparison of case merging and case splitting for symbolic execution. Such a comparison appears in the next chapter.

5.6. Symbolic Execution Summary

The last three chapters have discussed techniques for symbolically executing an event-driven simulation. The discussion has followed this outline:

Define a model for event-driven simulation. The model consists of concurrent processes which communicate with each other only through global variables in a controlled fashion. This protocol is defined so that the results of a simulation are independent of the interleaving of processes in the simulation. This follows the philosophy of SABLE.

Since simulation results are independent of process interleaving, we choose an interleaving in which processes do not overlap. Then we *may* consider symbolic execution of the simulation one process at a time. For some processes, a simple substitution of non-selective trace for selective trace eliminates case splitting without changing the functionality of the process. For more complex processes, the general case merging analyses from this chapter yield equivalent processes which do not introduce case splits when symbolically simulated.

In a symbolic simulation, the process created by a case merging analysis typically generates more complex symbolic expressions than those generated in a case splitting simulation. The next chapter discusses this problem further.

Chapter 6
Symbolic Execution Examples

Case merging seeks to improve the performance of symbolic execution by combining the results of all possible paths into a single execution. However, it typically does so at the cost of generating symbolic results far more complex than those produced in a case splitting symbolic execution. It is possible that the increased expression complexity can more than negate the reduction in the number of case analyses. This is true because of the difficulty of analyzing symbolic expressions.

This chapter presents qualitative criteria that suggest when case merging may be an effective approach for reducing the cost of a symbolic execution analysis. Section 6.1 discusses the complexity of generating and analyzing symbolic expressions. Section 6.2 discusses the complexity of case splitting and case merging symbolic executions.

Section 6.3 presents one example for which case splitting is preferable to case merging. This example includes two independent descriptions related by a simulation relation, as described in Section 2.2 for microcode verification. The complete results of the case splitting and case merging executions of the two descriptions appear in Appendix C.

Section 6.4 presents a second example for which case merging is preferable. This example also includes two descriptions plus the simulation relation.

6.1. Symbolic Expression Complexity

The factor favoring case merging over case splitting in symbolic execution is the reduction in the total number of paths. The factor favoring case splitting is the increased symbolic expression complexity introduced by case merging. Other factors remaining equal, we expect the computation time to increase linearly as the number of paths. The cost of the increased expression complexity in the analysis of the case merging results is more difficult to evaluate.

The fundamental operation in the analysis of the results of symbolic executions is the test that a Boolean assertion is identically true. We now consider the cost of this operation for simple Boolean expressions, universally quantified and not including arithmetic quantities or operations. Previously, we have not distinguished between a function and its symbolic representation. Now, we will let "string" denote the symbolic representation of a function.

Let $\{0,1\}^n$ denote the set of n-bit vectors. Then

$f: \{0,1\}^n \rightarrow \{0,1\}$

is a Boolean function of n Boolean variables. Let A^* be the set of finite-length strings using the alphabet A. The following result indicates the potential for complex strings representing Boolean functions.

Theorem 6.1: Given relation ***Rep***,

$Rep \subseteq (\{0,1\}^n \rightarrow \{0,1\}) \times A^*$,

with the following properties:

$\forall f1, f2 \in (\{0,1\}^n \rightarrow \{0,1\})$ and $s \in A^*$: $((f1\ Rep\ s)$ and $(f2\ Rep\ s)) \supset (f1 = f2)$,

$\forall f \in (\{0,1\}^n \rightarrow \{0,1\})$: $\exists s \in A^* \mid f\ Rep\ s$.

Let A^*_{Rep} denote the set of strings s in A^* such that s is the shortest string for which $f\ Rep\ s$ for some function f, and let $\alpha \geq 2$ be the number of symbols in A. Then the fraction of strings in A^*_{Rep} with length less than

$$\lceil (2^n / lg\,\alpha) \rceil - i - 1$$

is less than α^{-i}, for integer $i \geq 0$. □

The strings in A^*_{Rep} are the shortest symbolic-expression representations of the Boolean functions. Theorem 6.1 says that the shortest strings denoting most functions will have length exponential on the number of Boolean variables n. For example, assume twenty Boolean variables, with functions represented by bit strings ($\alpha = 2$). All the distinct functions could be represented by distinct bit strings of length 1048576 ($= 2^{20}$). But in any "compact" representation strategy, the fraction of *shortest* strings with length less than one million bits (about 95% of 1048576) must be less than 3×10^{-14623} ($= \alpha^{-i} = 2^{-48575}$)!

The proof of this theorem will not be given here. It is a simple counting argument identical to that used to prove that most bit strings have a high Kolmogorov complexity, a result stated by Martin-lof [Ma66].

We can try several approaches to keep string length to a minimum. One is to use a notation that is fairly compact for most functions actually computed. We can also attempt simplification to make the strings less complex.

However, testing that an assertion is identically true using a compact expression notation is likely to be np-hard. This claim is motivated by the proof that the satisfiability problem for Boolean functions represented in conjunctive normal form is np-complete. (See for example the text of Aho et al [AH74].) The proof generates a string of polynomial length representing a particular function. Such a string is most likely to exist in expression notations which strive to be compact.

Furthermore, regardless of any clever compact encoding or simplification, Theorem 6.1 ensures that in the worst case, the string length will be exponential on the number of variables n. This means that the worst case time to check that a function is identically true is an exponential of an exponential of n (assuming the np-complete problem cannot be solved in less than exponential time)!

A second approach, which may be used in conjunction with the first, is to enforce the use of a canonical representation, which ensures that each function has a unique representation. That is, require relation *Rep* in Theorem 6.1 to be a function. The test that two strings denote the same function becomes a simple check that the strings themselves are identical. This takes time proportional to the length of the shorter string, which is exponential on the number of variables n in the worst case.

An enhancement to this approach is to enter the strings in a hashing table, so that identical strings actually occupy the same storage. The test that two strings denote the same function is then a check that the two strings are in fact the same string, an O(1) pointer comparison.

The hidden cost in the use of a canonical form is the cost of computing the canonical representation for a given function. This is np-hard for compact canonical forms, since the *following procedure tests* whether an expression is satisfiable, an np-complete problem for compact expression notations:

1. Given an arbitrary expression, compute its canonical representation.
2. Test whether or not the canonical representation is *equal to the* canonical representation for *FALSE*.

If a hashing table is used, *then* strings hashing to the same location must be compared, taking time exponential on n in the worst case. Then the use of canonical representations and hashing does not remove the np-hardness associated with compact notations or the worst-case exponential length of strings, although it may otherwise improve the performance.

6.2. Case Splitting and Case Merging Complexity

The previous section *discussed the* complexity of Boolean expressions of n Boolean variables. Now consider a case splitting symbolic execution that uses n symbolic variables, all Boolean. In the extreme case, this execution could *split into* 2^n separate paths. In that event, each path corresponds to a distinct assignment of constant values *to the* symbolic variables. This corresponds to the exhaustive conventional execution of the program for all possible values of the n variables.

In conventional executions, all variables have constant values. The analysis of the results for each case is then $O(1)$ (independent of n). The total complexity of the exhaustive execution is $O(2^n)$.

If an actual symbolic *execution split into* 2^n *paths, the case* splitting may require some overhead to derive the constant variable values associated with each path. Alternately, the symbolic execution might split into few paths but generate complex symbolic strings. In these cases, the total computational complexity could exceed $O(2^n)$.

Conversely, *the case* splitting symbolic execution may split into few paths while generating simple symbolic strings. In such cases, the total computational complexity would be less than $O(2^n)$.

In case merging symbolic executions, we can use a compact notation in an effort to make "real" *strings (strings likely to be generated)* be of length polynomial on n. Comparison of these strings will be np-hard, but we can use heuristics that strive for polynomial performance in "real" problems.

If these heuristics are successful, then the total computational complexity of a case merging analysis will be a polynomial of n. Also, if most variables are modified *only in a few* basic blocks, then dataflow analysis can substantially simplify the symbolic expression computation. But in the worst case, the complexity will be an exponential of an exponential *of* n*, much worse than the* $O(2^n)$ complexity noted above for case splitting.

The large possible variations in the computational complexity of case splitting and case merging symbolic analyses prevent us from giving a simple quantitative criterion for preferring one method over the other in *specific* cases. However, the following sections present two examples which illustrate some possible qualitative criteria.

6.3. Blackjack Example

6.3.1. Blackjack Descriptions

Figure 6-1 gives the specification for the first example, a machine that plays Blackjack by dealer's rules. This description is adapted from a DDL description by Dietmeyer and Duley [DD75]. Call the description P_s.

```
(* Blackjack machine specification *)
var         Hit, Broke, Stand, FF: Boolean;
            Score, CardBuf: Integer;
input       YCrd: Boolean;
            Value: Integer;
begin
A: Hit := FALSE;    Broke := FALSE; Stand := FALSE; XA:
   Score := 0;      FF := FALSE;
B: Hit := TRUE;                                     XB:
   CardBuf := Value;
   if YCrd then goto C else goto B;
C: Hit := FALSE;                                    XC:
   if YCrd then goto C else goto D;
D: Score := Score + CardBuf;
   if (CardBuf ≠ 1) or FF then goto F else goto E;
E: FF := TRUE;    CardBuf := 10;    goto D;
F: if Score < 17 then goto B else goto G;
G: if Score < 22 then goto K else goto H;
H: CardBuf := -10;
   if FF then begin FF := FALSE; goto D end
      else goto J
J: Broke := TRUE;                                   XJ:
   if YCrd then goto A else goto J;
K: Stand := TRUE;                                   XK:
   if YCrd then goto A else goto K
end; (* Blackjack machine *)
```

Figure 6-1: Blackjack finite state machine specification

Figure 6-2 gives a proposed design, P_i, for this machine. To obtain P_i, we

interpret P_s as a finite state machine, with each **goto** representing a synchronous transfer to the next state. We assume that assignments to *Hit*, *Broke*, and *Stand* occur immediately upon entering the state, while all remaining assignments in the state do not occur until the machine exits the state. P_i then uses JK flip-flops to encode the state, combining states *F* and *G* into a single state.

6.3.2. Simulation Relation

Figure 6-3 is the simulation relation which describes how P_s and P_i should correspond to each other. For Boolean variables, it uses $0 \equiv FALSE$ and $1 \equiv TRUE$. Subscripts "s" and "i" denote variables in P_s and P_i, respectively. To denote that a particular assignment of values to variables satisfies the simulation relation, we write

$$r_s \, R_{si} \, r_i \quad .$$

where r_s (r_i) is the assignment of values for P_s (P_i). Formally, the simulation relation is R_{si}, the set of all value assignments $\langle r_s, r_i \rangle$ that satisfy the condition in Figure 6-3. We will say that a given value assignment r_s for P_s is a stopping point if there exists a value assignment r_i for P_i such that $r_s \, R_{si} \, r_i$. The same terminology applies for value assignments r_i for P_i.

Now let $F_{s/i}$ be the function that describes the action of P_s mapping one stopping point r_s to another. That is, initialize P_s by a value assignment r_s that is a stopping point. Execute P_s until it reaches a new stopping point s_s. Then by definition $s_s = F_{s/i}(r_s)$. $F_{i/s}$ is the corresponding function for P_i.

We now claim that P_i is a correct implementation of P_s by virtue of Milner's criterion [Mi71],

```
(* Blackjack machine design *)
var       Hit, Broke, Stand, Q4, Q5, Q6, FF: Boolean;
          J1, K1, J2, K2, J3, K3: Boolean;
          J4, K4, J5, K5, J6, K6, JFF, KFF: Boolean;
          Score, CardBuf: Integer;
input     YCrd: Boolean;
          Value: Integer;
begin
while TRUE do begin
    J1 := not (Broke or Stand or Q4 or Q6) or
          Q5 and not Q6 and (Score < 17);
    K1 := YCrd;
    J2 := Q4 and not Q5 and not FF;
    K2 := YCrd;
    J3 := Q5 and not Q6 and (17 ≤ Score < 22);
    K3 := YCrd;
    J4 := Q5;
    K4 := not Q5 or Q6 or (Score < 22);
    J5 := Q6 and not YCrd or Q4 and FF;
    K5 := Q4 and not Q6;
    J6 := Hit and YCrd or Q4 and not Q5 and FF;
    K6 := ((CardBuf ≠ 1) or FF) and not Q4 and Q5;
    JFF := Q4 and Q6;
    KFF := not (Hit or Stand or Q5 or Q6);
    Score := if not (Hit or Broke or Stand or Q4 or Q6) then 0 else
             if (not Q4 and Q5) then Score + CardBuf else Score;
    CardBuf := if Hit then Value else
               if (Q4 and Q6) then 10 else
               if (Q4 and not Q5) then -10 else CardBuf;
    Hit := J1 and not Hit or not K1 and Hit;
    Broke := J2 and not Broke or not K2 and Broke;
    Stand := J3 and not Stand or not K3 and Stand;
    Q4 := J4 and not Q4 or not K4 and Q4;
    Q5 := J5 and not Q5 or not K5 and Q5;
    Q6 := J6 and not Q6 or not K6 and Q6;
    FF := JFF and not FF or not KFF and FF
    end
end; (* Blackjack machine *)
```

Figure 6-2: Blackjack machine implementation

(* "@" denotes concatenation

$Lc \equiv$ location counter

$State_i \equiv Hit_i \,@\, Broke_i \,@\, Stand_i \,@\, Q4_i \,@\, Q5_i \,@\, Q6_i$ *)

$(Hit_s = Hit_i)$ and $(Broke_s = Broke_i)$ and $(Stand_s = Stand_i)$ and
$(FF_s = FF_i)$ and $(Score_s = Score_i)$ and $(CardBuf_s = CardBuf_i)$ and
$(YCrd_s = YCrd_i)$ and $(Value_s = Value_i)$ and
$(LC_i = \text{top})$ and
($((Lc_s = XA)$ and $(State_i = 000000))$ or
$((Lc_s = XB)$ and $(State_i = 100000))$ or
$((Lc_s = XC)$ and $(State_i = 000001))$ or
$((Lc_s = D)$ and $(State_i = 000011))$ or
$((Lc_s = E)$ and $(State_i = 000111))$ or
$((Lc_s = F)$ and $(State_i = 000110))$ or
$((Lc_s = H)$ and $(State_i = 000100))$ or
$((Lc_s = XJ)$ and $(State_i = 010000)$ and $(FF_i = 0))$ or
$((Lc_s = XK)$ and $(State_i = 001000))$)

Figure 6-3: Simulation relation for Blackjack descriptions

$\forall$ value assignments r_s for P_s and r_i for P_i:

$$(r_s \, R_{si} \, r_i) \supset (F_{s/i}(r_s) \, R_{si} \, F_{i/s}(r_i)) \qquad . \qquad (6.1)$$

The next chapter will consider criteria for correctness in detail; for now, the primary concern is the use of symbolic execution to test Eq. (6.1).

Symbolic execution helps verify Eq. (6.1) by testing many possible cases in parallel. We may use either case splitting or case merging to handle the conditional branches that arise during this verification. Appendix C outlines the symbolic execution of both descriptions using both case splitting and case merging. The appendix lists the complete results of such executions. We now consider the utility of case merging vs. case splitting in this example.

6.3.3. Symbolic Case Splitting Results

Figure 6-4 shows the results of the case splitting symbolic execution for a single case. This path is typical of the seventeen cases. (Results for all cases appear in Appendix C.) Two facts are apparent: (i) The symbolic values of the variables are very simple; in most cases, just symbolic variables or constants. (ii) The test that these results satisfy the simulation relation is trivial.

= =

(* "@" denotes concatenation *)

$State_s \equiv Lc_s : Hit_s \text{ @ } Broke_s \text{ @ } Stand_s$

$State_i \equiv Hit_i \text{ @ } Broke_i \text{ @ } Stand_i \text{ @ } Q4_i \text{ @ } Q5_i \text{ @ } Q6_i$

= =

Pc: $(c_0 \neq 1)$ or f_0

Initial:	$State_s = D{:}000$	$FF_s = f_0$	$Score_s = s_0$	$CardBuf_s = c_0$
Final:	$State_s = F{:}000$	$FF_s = f_0$	$Score_s = s_0 + c_0$	$CardBuf_s = c_0$

Initial:	$State_i = 000011$	$FF_i = f_0$	$Score_i = s_0$	$CardBuf_i = c_0$
Final:	$State_i = 000110$	$FF_i = f_0$	$Score_i = s_0 + c_0$	$CardBuf_i = c_0$

Figure 6-4: Case splitting results for one path

The second observation follows not only because of the first, but also because the initial value assignments r_s and r_i for each path guarantee that we have

$\forall$ value assignments r_s and r_i represented

by this symbolic execution: $r_s \, R_{si} \, r_i$.

Therefore, we need not prove Eq. (6.1):

$\forall$ represented value assignments r_s and r_i:

$$(r_s \, R_{si} \, r_i) \supset (F_{s/i}(r_s) \, R_{si} \, F_{i/s}(r_i)) \quad . \tag{6.1}$$

Rather, we need only show the simpler condition, Eq. (6.2):

$\forall$ represented value assignments r_s and r_i:

$$F_{s/i}(r_s) \, R_{si} \, F_{i/s}(r_i) \quad . \tag{6.2}$$

As noted above, this test is easy for the 17 paths in the case splitting symbolic execution.

Overall, we have the following statistics for P_s: P_s has 16 basic blocks. This happens to be the number of labels in the Blackjack description (15) plus the number of then-else clauses containing more than just a goto (one). In the symbolic execution of the 17 paths by case splitting, one block is executed three times, no other block is executed more than twice, and the average number of executions of a basic block is less than 1.4. The simulation state must be saved and later restored eight times due to case splits.

Description P_i has a single basic block. It is executed 17 times, once for each path from P_s.

6.3.4. Symbolic Case Merging Results

Now consider the results of the case merging executions. Figure 6-5 shows the initialization of some pertinent variables and the symbolic results for variable *Hit*. (In Appendix C, the program counter of P_s is represented in a distributed fashion by array *LPc*.) We assume that the initial value assignments satisfy R_{si}.

Then, for the final results to satisfy R_{si}, the final values of Hit_s and Hit_i should be equal.

= =

$State_s \equiv Lc_s : Hit_s \ @\ Broke_s \ @\ Stand_s$

$State_i \equiv Hit_i \ @\ Broke_i \ @\ Stand_i \ @\ Q4_i \ @\ Q5_i \ @\ Q6_i$

= =

Initial: $State_s = p_0 : h_0 b_0 t_0$ $\quad FF_s = f_0 \quad Score_s = s_0$

$CardBuf_s = c_0 \quad YCrd_s = y_0$

Final:

$Hit_s = (p_0 = XA)$ or $((p_0 = XB)$ and not $y_0)$ or $((p_0 = F)$ and $(s_0 < 17))$ or
$(h_0$ and not $(y_0$ and $p_0 \in \{XB, XC, XJ, XK\}))$

Initial: $State_i = h_0 b_0 t_0 u_0 w_0 x_0$ $\quad FF_i = f_0 \quad Score_i = s_0$

$CardBuf_i = c_0 \quad YCrd_i = y_0$

Final:

$Hit_i = (($not $(b_0$ or t_0 or u_0 or $x_0)$ or
w_0 and not x_0 and $(s_0 < 17))$ and not $h_0)$ or
not y_0 and h_0

Figure 6-5: Case merging results for one variable

Figure 6-6 shows the final symbolic value of Lc_s in the case merging symbolic execution of P_s. We make two observations on the case merging results.

First, the final case merging symbolic expressions are much more complex than those from the case splitting executions. The expressions for *Hit* are typical. The final value of Lc_s is the most complex symbolic expression generated. Note that the condition on Lc_s in R_{si} is also the most complex condition in R_{si}, so the verification of the implementation of Lc_s will be especially complex.

$Lc_s =$ if $((p_0 = XJ)$ and $y_0)$ or $((p_0 = XK)$ and $y_0)$ then XA else
if $(p_0 = XA)$ or $((p_0 = XB)$ and not $y_0)$ or
$((p_0 = F)$ and $(s_0 < 17))$ then XB else
if $((p_0 = XB)$ and $y_0)$ or $((p_0 = XC)$ and $y_0)$ then XC else
if $((p_0 = XC)$ and not $y_0)$ or $(p_0 = E)$ or $((p_0 = H)$ and $f_0)$ then D else
if $(p_0 = D)$ and $(c_0 = 1)$ and not f_0 then E else
if $(p_0 = D)$ and $((c_0 \neq 1)$ or $f_0)$ then F else
if $(p_0 = F)$ and $(s_0 \geq 17)$ and $(s_0 \geq 22)$ then H else
if $((p_0 = H)$ and not $f_0)$ or $((p_0 = XJ)$ and not $y_0)$ then XJ else
if $((p_0 = F)$ and $(s_0 \geq 17)$ and $(s_0 < 22))$ or
$((p_0 = XK)$ and not $y_0)$ then XK

Figure 6-6: Final case merging symbolic program counter

Second, we cannot use Eq. (6.2) to verify these symbolic results. This holds partially because the final value of Hit_i uses symbolic variables u_0, w_0, and x_0. These appear only in the execution of P_i and are related to other variables by the left side of Eq. (6.1). However, this does not completely explain the problem.

R_{si} implies the following equalities for u_0, w_0, and x_0.

$u_0 \equiv p_0 \in \{E, F, H\}$,

$w_0 \equiv p_0 \in \{D, E, F\}$,

$x_0 \equiv p_0 \in \{XC, D, E\}$.

If we substitute these values for u_0, w_0, and x_0, then the symbolic executions of P_s and P_i both use the same set of symbolic variables, yet their equivalence still cannot be proven by Eq. (6.2).

6.3.5. Conclusion to First Example

In general, it is not necessarily true that a case splitting simulation will allow the use of Eq. (6.2). Conversely, in the Blackjack example, we could use Eq. (6.2) to verify the case merging results if we substituted expressions for h_0, b_0, and t_0 as well as for u_0, w_0, and x_0. However, our expectation is that Eq. (6.2) is less likely to be valid in case merging analyses than it is in case splitting analyses. This aggravates the problem of increased expression complexity. For the Blackjack machine, a case splitting symbolic execution and analysis is likely to be less expensive than the corresponding case merging analysis.

What properties of the Blackjack example make case merging undesirable? The outstanding characteristic is the highly detailed simulation relation R_{si}. This is partially responsible for the following properties:

1. A minimum of case splitting occurs in the symbolic execution starting from any stopping point. In only one case does the execution split into as many as three paths (namely, starting from label *F*).
2. Each path is short and simple. Most paths modify few variables.

This in turn simplifies the case splitting symbolic results.

On the other hand, the high level of detail in R_{si} causes the case merging results to be complex, with the computation of complicated conditional expressions. A dataflow analysis allows most of the symbolic values in the Blackjack machine to be simplified, but the value of Lc_s remains complex. The condition on Lc_s in R_{si} is also complex. Finally, Eq. (6.2) is less likely to be valid in the case merging analysis when R_{si} is highly detailed.

6.4. Associative Memory Example

The above example suggests that case merging is not beneficial when the simulation relation is detailed, including a stopping point in most of the basic blocks of a description. Conversely, we might expect that case merging may be advantageous if we have a "sparse" simulation relation. This section considers a simple example to illustrate this point.

Figure 6-7 shows a simple associative memory which finds the appropriate address by a linear search. Figure 6-8 shows a similar description which specifies a tree structure for extracting the correct address. The simulation relation in Figure 6-9 defines the correspondence between the two versions. The subscripts "*s*" and "*i*" refer to Figures 6-7 and 6-8, respectively.

```
var M: array [0..3] of Integer;
    Data: Integer;
    Found: Boolean;
    Addr: 0..3;
begin (* CAM *)
Found := TRUE;
Addr := 0;
(**) if M[0] = Data then (* nothing *)
else if M[1] = Data then Addr := 1
else if M[2] = Data then Addr := 2
else if M[3] = Data then Addr := 3
else Found := FALSE
end; (* CAM *)
```

Figure 6-7: Associative memory description

The comparison of these descriptions by symbolic execution is similar to that for the Blackjack example. One difference is that only the stopping points with Lc = **top** are used as the starting point of a symbolic execution. This is because

```
var M: array [0..3] of Integer;
    Data: Integer;
    F1, F2, Found: Boolean;
    A1, A2, Addr: 0..3;
begin (* CAM *)
F1 := TRUE;
(**) if M[0] = Data then A1 := 0
 else if M[1] = Data then A1 := 1
 else F1 := FALSE;
F2 := TRUE;
A2 := 3;
(**) if M[2] = Data then A2 := 2
 else if M[3] ≠ Data then F2 := FALSE;
if F1 then Addr := A1 else Addr := A2;
Found := F1 or F2
end; (* CAM *)
```

Figure 6-8: Second associative memory description

```
(    (Lc_s = top) and (Lc_i = top) and
     (Data_s = Data_i) and (M_s = M_i)        ) or
(    (Lc_s = bottom) and (Lc_i = bottom) and
     (    (not Found_s and not Found_i) or
          (Found_s and Found_i and (Addr_s = Addr_i))      )      )
```

Figure 6-9: Simulation relation for associative memory

Lc = **bottom** represents the end of the procedure; the procedure is not an infinite loop as is the Blackjack machine. Figure 6-10 shows the results of case splitting simulations for the two descriptions.

In the symbolic execution of Figure 6-7, variables Lc_s, M_s, $Data_s$, $Found_s$, and $Addr_s$ have initial values **top**, m_0, d_0, f_0, and a_0, respectively. In Figure 6-8, variables Lc_i, M_i, $Data_i$, $Found_i$, $Addr_i$, $F1_i$, $F2_i$, $A1_i$, and $A2_i$ have initial values **top**, m_0, d_0, f_3, a_3, f_1, f_2, a_1, and a_2. The simulation relation implies the equality of the initial values of *Lc*, *M*, and *Data* in the two simulations.

Pc	$Found_i$	$Addr_i$	$F1_i$	$F2_i$	$A1_i$	$A2_i$	$Found_s$	$Addr_s$
$e_0 \; e_2$	T	0	T	T	0	2	T	0
$e_0 \; n_2 e_3$	T	0	T	T	0	3	T	0
$e_0 \; n_2 n_3$	T	0	T	F	0	3	T	0
$n_0 e_1 e_2$	T	1	T	T	1	2	T	1
$n_0 e_1 n_2 e_3$	T	1	T	T	1	3	T	1
$n_0 e_1 n_2 n_3$	T	1	T	F	1	3	T	1
$n_0 n_1 e_2$	T	2	F	T	a_1	2	T	2
$n_0 n_1 n_2 e_3$	T	3	F	T	a_1	3	T	3
$n_0 n_1 n_2 n_3$	F	3	F	F	a_1	3	F	0

Figure 6-10: Case splitting symbolic execution results

The first column in Figure 6-10, *Pc*, gives the final path condition for the nine paths in Figure 6-8. In this column, e_j denotes $m_0[j] = d_0$ and n_j denotes $m_0[j] \neq d_0$. The remaining columns specify final symbolic values of variables in the two simulations. The results satisfy the simulation relation in all nine paths. In each case, the test of the simulation relation is trivial; *Found* and *Addr* have constant values.

Figures 6-11 and 6-12 gives the symbolic results of case merging simulations for Figures 6-7 and 6-8. These results also verify that the simulation relation is satisfied.

This example differs from the Blackjack example in that the simulation relation is not detailed. It includes only one stopping point (excluding the exit point) in each program. Program P_1 uses a tree structure that, when generalized, splits the case splitting symbolic execution into a number of paths exponential on the size *n* of *M*, into $3^{n/2}$ paths.

```
$Found_s = e_0$ or $n_0e_1$ or $n_0n_1e_2$ or $n_0n_1n_2e_3$
        $= e_0$ or $e_1$ or $e_2$ or $e_3$;
$Addr_s =$ if $e_0$ then 0 else
        if $n_0e_1$ then 1 else
        if $n_0n_1e_2$ then 2 else
        if $n_0n_1n_2e_3$ then 3 else
        if $n_0n_1n_2n_3$ then 0;
```

Figure 6-11: Case merging results for Figure 6-7

By contrast, with case merging we could use a simple canonical representation in which the expression complexity grows linearly as n. The total cost of the case merging analysis may be polynomial, depending on the cost of simplification. Also, Eq. (6.2) is valid with case merging here, even though this was not true for the Blackjack example.

This example does not represent a realistic application of symbolic execution. Due to the regular structure in both Figures 6-7 and 6-8, a program proof by induction on n would be preferable to a proof by symbolic execution. Nonetheless, this example illustrates characteristics which motivate a preference for case merging over case splitting: a simulation relation with little detail (few stopping points), in which the number of paths grows exponentially while the symbolic expression complexity grows linearly.

6.5. Conclusion

This chapter has presented two examples to illustrate the issues in the comparison of case splitting and case merging symbolic executions. The computational complexity of these two approaches varies over such a wide range

```
$F1_i =$ not ($n_0$ and $n_1$) $= e_0$ or $e_1$;
$A1_i =$ if $e_0$ then 0 else
       if $n_0e_1$ then 1 else
       if $n_0n_1$ then $a_1$;
$F2_i =$ not ($n_2$ and $n_3$) $= e_2$ or $e_3$;
$A2_i =$ if $e_2$ then 2 else
       if $n_2$ then 3;
$Addr_i =$ if $e_0$ or $e_1$ then (if $e_0$ then 0 else
                              if $n_0e_1$ then 1 else
                              if $n_0n_1$ then $a_1$) else
          if $n_0n_1$ then (if $e_2$ then 2 else
                           if $n_2$ then 3)
        $=$ if ($e_0$ or $e_1$) and $e_0$ then 0 else
          if ($e_0$ or $e_1$) and $n_0e_1$ then 1 else
          if ($e_0$ or $e_1$) and $n_0n_1$ then $a_1$ else
          if $n_0n_1e_2$ then 2 else
          if $n_0n_1n_2$ then 3
        $=$ if $e_0$ then 0 else
          if $n_0e_1$ then 1 else
          if $n_0n_1e_2$ then 2 else
          if $n_0n_1n_2$ then 3;
$Found_i = e_0$ or $e_1$ or $e_2$ or $e_3$;
```

Figure 6-12: Case merging results for Figure 6-8

that we cannot make general quantitative statements about the superiority of one approach or the other. However, the examples illustrated qualitative properties that might suggest a preference in certain cases.

In the Blackjack example, the simulation relation was highly detailed, including many different stopping points in P_s. Case splitting symbolic executions split into few paths and allowed the use of Eq. (6.2). But with case merging, the detailed simulation relation required the proof of Eq. (6.1) rather than the simpler Eq. (6.2). The symbolic expression complexity for Lc_s and Lc_t was linear on the number of paths in the case merging analysis. Case splitting was preferred in this case.

In the associative memory example, the simulation relation had little detail, including only one stopping point (excluding the exit point). The case splitting executions split into an exponential number of paths. The case merging execution results grew in complexity as the log of the number of paths. Also, Eq. (6.2) was valid in this case merging analysis. For this example, case merging was preferable.

Implementation and experimentation with high-quality case splitting and case merging symbolic execution systems is desirable to gain more insight into the relative strengths of the two approaches, particularly when applied to event-driven simulations (an application not considered in these examples). In addition to the above considerations, case merging might be effective if we can guarantee an acceptable upper bound on the complexity of the symbolic expressions generated, for example by limiting the number of symbolic inputs in the program executions.

Chapter 7
Criteria for Consistency

7.1. Overview

A criterion for consistency is a test by which we determine whether a proposed hardware design is a correct implementation of the given specification. The design and specification are expressed by hardware descriptions.

We will call hardware descriptions "programs" and represent them by graphs. Note however that these "programs" represent hardware, not software, and this may affect verification. For example, Milner was concerned with software, and some of his work stressed the final state of a program. But a correct hardware description might never terminate, so that a "final state" is of no importance.

Milner's criterion for consistency uses a restrictive program model which is not conveniently applicable for hardware design verification (Section 7.3). Brand's criterion verifies clearly incorrect designs (Section 7.4). Two issues in the attempt to improve these criteria are

1. characteristics of the program model (Section 7.5), and
2. intuitive understanding of correctness (Sections 7.6 - 7.8)

A single, simple criterion can be found which is appropriate for a variety of program

models (Section 7.8). This test captures intuitive notions of correctness, but is tedious to check by symbolic execution. A second criterion is easier to test, but more complex than the first (Section 7.9). Given two deterministic programs which satisfy the second definition of consistency, we can show that they also satisfy the first criterion under a modified (but still "reasonable") simulation relation (Section 7.10).

It is desirable to apply the criterion for consistency in a hierarchical, partitioned fashion. Hierarchical verification using the criterion from Section 7.8 is possible (Section 7.11). We know how to allow partitioned verification (1) only for an inappropriate criterion for consistency, (2) only for a nondeterministic program model, and (3) only assuming a property of the program model which is unreasonable for some hardware models (Brand's technical report [Br78]).

A gap remains which must be filled to create a unified theory for the criterion for consistency of programs. Section 7.9 presents an extended criterion which may be checked by symbolic execution. However, we have proven hierarchical, partitioned verification only for other definitions. Two approaches for closing this gap are the following:

- Prove hierarchical, partitioned verification for the extended criterion.
- Prove partitioned verification for the simpler criterion from Section 7.8. Then prove a correspondence between this definition and the extended definition. The results for hierarchical, partitioned verification with the simple criterion may also hold for the extended criterion, according to this correspondence.

Section 7.10 shows a correspondence between the two definitions, but only for deterministic program models; Brand demonstrated partitioned verification only for nondeterministic models.

7.2. Notation

This chapter will use graph models to represent programs. The discussion will use terms from graph theory. A "vertex" represents a program state, while an edge represents an action of the program in changing the program state.

When used as subscripts, the letters "s", "i", and "l" will denote particular programs. In the first parts of the chapter, we use only "s" and "i", denoting *specification* and *implementation*. Later we add "l", and the three letters denote *specification*, *intermediate*, and *low level* in a hierarchical design.

When used as subscripts, "a" and "b" will denote unspecified programs. That is, program *a* might be program *s*, *i*, or *l*.

Appendix A describes notational conventions used here for relations.

7.3. Simulation between Deterministic Programs

This section reviews Milner's discussion of algebraic simulation between programs [Mi71]. We will omit certain details not of concern here.

Milner uses an abstract graph-like representation for a program. The set D_a (a vertex set) is the set of all possible states in the program. A state is an assignment of values to the program variables, including the contents of the stack and the location

counter. Relation $F_a \subseteq D_a \times D_a$ (an edge set) specifies how the program maps any state into a successor state. Milner requires that the program be deterministic; that is, that F_a be a function. A program P_a (a graph) is then the ordered pair $\langle D_a, F_a \rangle$.

Figure 7-1 illustrates the verification problem. Programs $P_s = \langle D_s, F_s \rangle$ and $P_i = \langle D_i, F_i \rangle$ are proposed specification and implementation programs. A relation $R_{ab} \subseteq D_a \times D_b$ specifies how vertices in D_b should correspond to vertices in D_a. In particular, $R_{si} \subseteq D_s \times D_i$, and $R_{is} = R_{si}^{-1}$. Relation R_{si} is usually called the "simulation relation." By definition, P_i is consistent with P_s under relation R_{si} if and only if

$$\forall \langle s,u \rangle \in D_s \times D_i \mid s\, R_{si}\, u:\ F_s(s)\, R_{si}\, F_i(u) \quad .$$

Milner expresses this, "R_{si} is a simulation of P_s by P_i," emphasizing the importance of relation R_{si} in the definition. Since F_s and F_i are functions, this criterion is equivalent to both of the following:

$$R_{si} F_i \subseteq F_s R_{si} \quad . \qquad (7.1)$$

$$R_{is} F_s \subseteq F_i R_{is} \quad . \qquad (7.2)$$

Note that F_s is required to be defined for all vertices in D_s. This disallows don't-care conditions that could lead to more optimal designs.

7.4. Simulation between Nondeterministic Programs

Brand modifies Milner's development in three ways [Br78]. First, Brand allows a program P_a to be nondeterministic; relation F_a need not be a function. Second, given P_a, P_b, and R_{ab}, he defines new program $P_{a/b}$. Note that $P_{a/b}$ is a version of P_a modified using R_{ba} ($= R_{ab}^{-1}$):

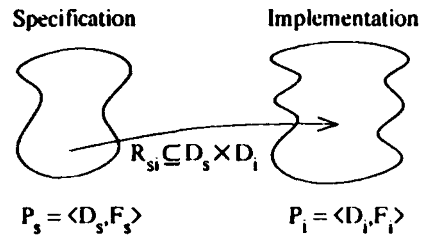

Specification
Implementation
$R_{si} \subseteq D_s \times D_i$
$P_s = \langle D_s, F_s \rangle$
$P_i = \langle D_i, F_i \rangle$

Figure 7-1: Functional verification problem

$$\begin{aligned}
&D_{a/b} = R_{ba}(D_b)\\
&F_{a/b} = \{\langle s,t\rangle \in D_{a/b} \times D_{a/b} \mid (\exists\, x_0, x_1, \ldots, x_n \in D_a\ (n \geq 1) \mid\\
&\qquad s = x_0\, F_a\, x_1\, F_a \ldots F_a\, x_n = t, \text{ and}\\
&\qquad \text{of the several } x_i, \text{ only } x_0 \text{ and } x_n \text{ are in } D_{a/b})\}\\
&P_{a/b} = \langle D_{a/b}, F_{a/b}\rangle \quad . \qquad (7.3)
\end{aligned}$$

This allows a coarse definition of relation R_{ab}. Third, he adds closure to Eq. (7.1) to obtain a new criterion for consistency using $P_{s/i}$ and $P_{i/s}$:

$$R_{si}F_{i/s}^{*} \subseteq F_{s/i}^{*}R_{si} \quad . \qquad (7.4)$$

This development leading to Eq. (7.4) is slightly different from Brand's, but the effective result is the same.

Eq. (7.4) has the unacceptable property of "verifying" the correctness of incorrect implementations, regardless of the choice of R_{si}. In fact, we now show that if program P_s is strongly connected, then an arbitrary implementation P_i satisfies the above criterion.

PROOF: Consider an arbitrary pair $\langle s,v\rangle \in R_{si}F_{i/s}^{*}$. It will suffice to show that this pair is in $F_{s/i}^{*}R_{si}$. From the definition of $D_{s/i}$, $D_{i/s}$, and $F_{i/s}$ in Eq. (7.3), s

must be in $D_{s/i}$ and v in $D_{i/s}$. Therefore there exists a vertex t in $D_{s/i}$ such that $t\, R_{si}\, v$. If P_s is strongly connected, then by the construction of $D_{s/i}$ and $F_{s/i}$ using Eq. (7.3), program $P_{s/i} = \langle D_{s/i}, F_{s/i}\rangle$ is also strongly connected. Hence $s\, F_{s/i}^{*}\, t$. Now $s\, F_{s/i}^{*}\, t\, R_{si}\, v$, or $\langle s,v\rangle \in F_{s/i}^{*}R_{si}$. QED.

In view of the comments on the modeling of changing inputs in Section 7.5 (later), it is unreasonable to expect that P_s will be strongly connected. However, the above proof is instructive. For every path in P_i, Eq. (7.4) ensures only that a path connecting corresponding points exists in P_s. It proves nothing about desirable properties of the path itself; that the path passes through appropriate intermediate states.

It is not surprising that some incorrect implementations can be shown correct for some choices of R_{si}. By both Eqs. (7.1) and (7.4), every implementation is consistent with the specification under the empty relation. However, the above result says that every (incorrect) design may satisfy the criterion for consistency for *every* choice of R_{si}.

This problem arises because of the presence of closure in Eq. (7.4). The justification for closure is as follows: Consider states s_n and t in $D_{s/i}$, and u and v in $D_{i/s}$, such that (Figure 7-2)

- $s_n\, R_{si}\, u\, F_{i/s}\, v$ and
- $s_n\, F_{s/i}\, t\, R_{si}\, v$.

Suppose state s_n is inside a tight loop not including state t, while u is not in any such loop. This might occur because the implementation P_i uses a more suitable operator so that the loop is not necessary. Then all the corresponding states s_i for the same location in every pass through the loop are also related to u by R_{si}. For each such state we have $s_i\ R_{si}\ u\ F_{i/s}\ v$. Figure 7-2 illustrates the situation. The relation $s_i\ F_{s/i}\ t\ R_{si}\ v$ does not hold, although $s_i\ F_{s/i}^*\ t\ R_{si}\ v$ is true. The removal of closure from the right hand side of Eq. (7.4) prevents the verification of a possibly correct implementation in this case. The argument for closure in the left hand side is similar.

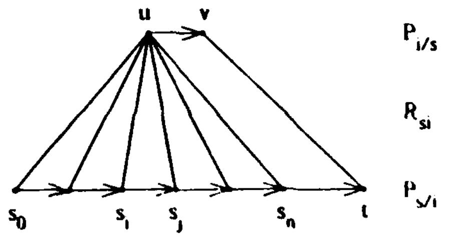


Figure 7-2: Example motivating use of closure

This argument assumes that states in the relation R_{si} are identified solely by their location counters, so that all states sharing the same location counter are related to the same states by R_{si}. This assumption is not necessary. The definition of states in the relation R_{si} may include values of variables (including the stack) as well as the location counter. Closure is not required for this purpose.

This refutation is tech[illegible]ally correct, but ignores a practical reason for desiring that states in relation R [illegible]e identified solely by their location counters. The practical consideration *involves the use of symbolic simulation* to compare program models. Section 7.9 will discuss this question in detail.

7.5. Program Models

Milner and Brand used different program models and proposed different criteria for consistency. The two approaches have different shortcomings. The next few sections will take a more general look at possible types of program models and consistency criteria.

This section identifies several types of program models which might be used to describe hardware. A later section will consider definitions of consistency for each of these models.

The models used by Milner and Brand differed in two ways. First, Milner required the program to be deterministic, while Brand allowed the model to be nondeterministic. Second, Milner required that all inputs be presented before program execution, *while Brand made no such restriction.*

A hardware component generally reads its inputs many times during its normal operation. These inputs usually are allowed to change many times. Therefore, if a program model is to be used to represent hardware, then it is not reasonable to require that all inputs be presented before the start of execution.

On first inspection, the use of a deterministic model appears to prohibit the modeling of a program with changing inputs. If inputs may change, then from the program's point of view, these changes are nondeterministic. It seems that the model for this program must be nondeterministic. However, this is true only if the model explicitly includes the random changing of inputs. A model might include only the actions taken by the program itself, not the random changes from the program's environment. In fact, we will always assume that the graph does not model changes to inputs by the environment. Then both deterministic and nondeterministic models can be used for modeling hardware.

Milner did not allow don't-cares in the deterministic program model; for every state, a next state had to be defined. Brand did not explicitly support don't-cares, although in a nondeterministic model, one can indicate a don't-care condition for state s by specifying every state as a possible successor to s. Depending on one's design philosophy, the use of don't-cares may or may not be desirable.

In a program graph model P_a, a vertex is a don't-care vertex if it has no successors. *In that case*, we assume that any vertex is allowed to be the successor. Any vertex which has one or more successors explicitly specified in the model is not a don't-care vertex; its next vertex must be one of the defined successor vertices. D_a^x is the set of don't-care vertices in D_a:

$$D_a^x = \{\text{vertices } s \in D_a \mid \neg(\exists t \mid s\ F_a\ t)\} \quad . \qquad (7.5)$$

This definition breaks down for the program $P_{a/b}$ constructed from Eq. (7.3). It is possible for a vertex s in $D_{a/b}$ to have no successor in $P_{a/b}$ although it does not lead to a don't-care vertex in P_a. For example, the only path from vertex s in P_a could be $s\ F_a\ t\ F_a\ t$, where t is not in $D_{a/b}$. In this case, $P_{a/b}$ fails to properly *characterize the behavior of* P_a at s. We can eliminate this anomaly by the following restriction.

Requirement 1: Every path in P_a leading from any vertex in $D_{a/b}$ must eventually reach either a vertex in $D_{a/b}$ or a don't-care vertex. □

We also assume the state space D_a is finite. Then Requirement 1 is satisfied if and

only if every loop in P_a that is reachable from a vertex in $D_{a/b}$ contains a vertex in $D_{a/b}$.

Even with this restriction, a possible anomaly still exists for nondeterministic models. Given vertex s in $D_{a/b}$, if a path in P_a from s reaches a don't-care vertex without passing through intervening vertices in $D_{a/b}$, then s should be considered a don't-care vertex in $P_{a/b}$. However, *if other paths from s reach vertices in $D_{a/b}$*, then s will have explicitly specified successors in $P_{a/b}$ (by Eq. (7.3)), and so will not be recognized as a don't-care vertex. We can handle this problem by modifying Eq. (7.3):

$$\begin{aligned}
&D_{a/b} = R_{ba}(D_b)\\
&F_{a/b} = \{\langle s,t\rangle \in D_{a/b} \times D_{a/b} \mid\\
&\qquad (\exists\, x_0, x_1, \dots, x_n \in D_a\ (n \geq 1) \mid\\
&\qquad\qquad s = x_0\ F_a\ x_1\ F_a \dots F_a\ x_n = t, \text{ and}\\
&\qquad\qquad \text{of the several } x_i, \text{ only } x_0 \text{ and } x_n \text{ are in } D_{a/b}) \text{ and}\\
&\qquad (\text{there do not exist } y_0, y_1, \dots, y_n \in D_a\ (n \geq 1) \mid\\
&\qquad\qquad s = y_0\ F_a\ y_1\ F_a \dots F_a\ y_n,\\
&\qquad\qquad y_n \text{ is a don't-care state, and}\\
&\qquad\qquad \text{of the several } y_i, \text{ only } y_0 \text{ is in } D_{a/b})\}\\
&P_{a/b} = \langle D_{a/b}, F_{a/b}\rangle \quad . \qquad\qquad (7.6)
\end{aligned}$$

Given Eq. (7.6) and Requirement 1 above, then vertex s is a don't-care vertex in $P_{a/b}$ if and only if $s\ F_a{}^*\ t$, where t is a don't-care vertex (possibly s) in P_a, and some path $s\ F_a{}^*\ t$ includes no vertices in $D_{a/b}$ except s. The definition of don't-care vertices is now the same for both P_a and $P_{a/b}$. A don't-care vertex is a vertex with no explicitly declared successors.

We will consider consistency criteria for four types of program models:

deterministic with don't-cares, deterministic without don't-cares, nondeterministic with don't-cares, and nondeterministic without don't-cares.

7.6. Consistency Criteria

Eqs. (7.1) and (7.2) may be applied to programs $P_{s/i}$ and $P_{i/s}$ defined by Eq. (7.6):

$$R_{si}F_{i/s} \subseteq F_{s/i}R_{si} \quad . \qquad (7.7)$$

$$R_{is}F_{s/i} \subseteq F_{i/s}R_{is} \quad . \qquad (7.8)$$

Eq. (7.7) specifies that the existence of an edge in the implementation $P_{i/s}$ implies the existence of a corresponding edge in the specification $P_{s/i}$. We will call criteria of this type "implied specifications." Eq. (7.8) specifies that the existence of an edge in the specification implies a corresponding edge in the implementation. Such criteria are "implied implementations."

The next section will consider the choice between an implied specification and an implied implementation. This section considers the selection of one implied specification from several simple alternatives. An analogous discussion applies to implied implementations.

Four simple implied specifications are as follows:

$$R_{si}F_{i/s}R_{is} \subseteq F_{s/i} \quad . \qquad (7.9)$$

$$R_{si}F_{i/s} \subseteq F_{s/i}R_{si} \quad . \qquad (7.7)$$

$$F_{i/s}R_{is} \subseteq R_{is}F_{s/i} \quad . \qquad (7.10)$$

$$F_{i/s} \subseteq R_{is}F_{s/i}R_{si} \quad . \qquad (7.11)$$

Consider an edge $\langle u,v \rangle$ in $P_{i/s}$. Eq. (7.9) requires that any vertex related to v by R_{is}

must be a successor of every vertex related to u by R_{is}. This is clearly unreasonable; Eq. (7.9) will not be considered further.

Given the edge $\langle u,v \rangle$ in $P_{i/s}$: Eq. (7.7) says that there is an edge $\langle s,t \rangle$ in $F_{s/i}$ corresponding to $\langle u,v \rangle$ for *every* vertex s in $D_{s/i}$ related to u by R_{si}. Eqs. (7.10) and (7.11) guarantee only that there is an edge $\langle s,t \rangle$ corresponding to $\langle u,v \rangle$ for *some* vertex s in $D_{s/i}$ related to u by R_{si}. The former requirement is preferable. It guarantees that the action of the implementation at state u is correct regardless of how it happens to be related to the specification during a particular execution of programs P_s and P_i.

One might argue that a corresponding edge $\langle s_j,t \rangle$ need not exist for a particular vertex s_j related to u by R_{si}, because we happen to know that P_s will never be at s_j when P_i is at u. But if this is the case, then $\langle s_j,u \rangle$ should not be in R_{si}. Alternately, we might happen to know that when P_s is at s_j, then we do not care what the next state is. But in this case, the problem is that the specification need not be implied by the implementation at this point. This consideration argues for the choice between Eqs. (7.7) and (7.8), not the choice between (7.7), (7.10), and (7.11).

Eq. (7.10) guarantees that there is an edge $\langle s,t \rangle$ in $F_{s/i}$ corresponding to $\langle u,v \rangle$ for every vertex t in $D_{s/i}$ related to v by R_{si}. This test is of little value. Given two corresponding states $t\ R_{si}\ v$, we are concerned not with how the programs P_s and P_i came to be in those states, but rather with the consistency of the future actions of the programs.

The above considerations motivate the use of Eq. (7.7) rather than (7.9), (7.10), or (7.11). This confirms the general form of the criteria for consistency used by both Milner and Brand.

7.7. Implied Implementation vs. Implied Specification

This section considers four program models, determining whether Eq. (7.7) or (7.8) is preferable for each case.

7.7.1. Deterministic Programs without Don't-Cares

If P_s and P_i are deterministic with no don't-cares, then Eq. (7.6) and Requirement 1 from Section 7.5 guarantee that $P_{s/i}$ and $P_{i/s}$ will also be deterministic with no don't-cares. Milner noted that Eqs. (7.7) and (7.8) are exactly equivalent for such programs. The choice of one or the other is arbitrary.

7.7.2. Deterministic Programs with Don't-Cares

Three issues arise in defining consistency for deterministic programs with don't-cares.

1. The specification should be completely implemented. This is ensured by Eq. (7.8) but not by Eq. (7.7). Brand claimed that one could overcome this problem with Eq. (7.7) by providing a sufficiently detailed relation R_{si} [Br78]. However, his discussion assumed that there were no don't-cares in the implementation. If this is not the case, then the detailed relation does not solve the problem.

2. Since don't-cares are allowed in the specification, we should not require that every action of the implementation be explicitly given in the specification. Eq. (7.7) makes this requirement, but Eq. (7.8) does not. The following example illustrates this point.

Figure 7-3 shows a graph representation of a specification program $P_{s/i}$. The letter inside each vertex is a name for the program state. The notation "i/xx" gives the system inputs and outputs associated with each state; "i" is the Boolean input while "xx" are the Boolean outputs. For example, the values 0/11 on state *i* indicates that the system enters state *i* from states *e* or *h* when the input is 0. (This is different from typical finite state machine graphs where the I/O designations are attached to edges, not vertices.)

Asterisks separate pairs of states which are identical except for the value of the input. Since the graph only models the behavior of the program itself, the edges ($F_{s/i}$) connect only states with identical inputs. However, in an actual execution, the environment may cause transfers between states separated by asterisks, for example from *g* to *h*. Hence, the model permits changing inputs, but $F_{s/i}$ does not explicitly represent the effects of changing inputs.

The designer may combine states *d* and *i* in the implementation, yielding program $P_{i/s}$ shown in Figure 7-4. The relations $F_{s/i}$ and $F_{i/s}$ appear in Figure 7-5, along with relation $R_{si} \subseteq D_{s/i} \times D_{i/s}$.

The four relations $(R_{si}F_{i/s})$, $(F_{s/i}R_{si})$, $(R_{is}F_{s/i})$, and $(F_{i/s}R_{is})$ appear in Figure 7-6. The relations satisfy Eq. (7.8), but not Eq. (7.7). In general, Eq. (7.7) does not allow incomplete specifications as in Figure 7-3. This makes optimizations awkward, requiring the modification of $P_{s/i}$ to include the optimizations performed.

3. The first two considerations motivate a preference for Eq. (7.8). The third

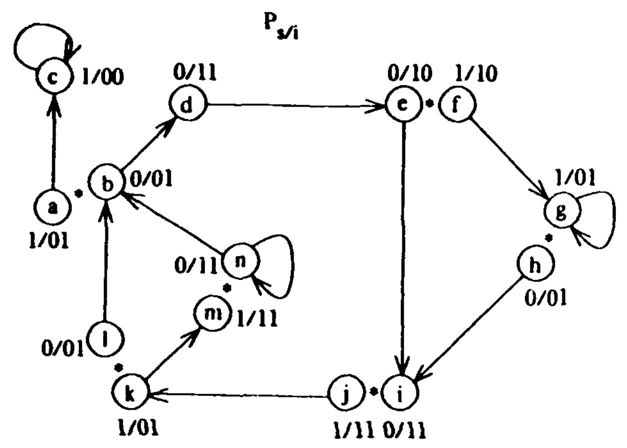


Figure 7-3: Example - Specification program

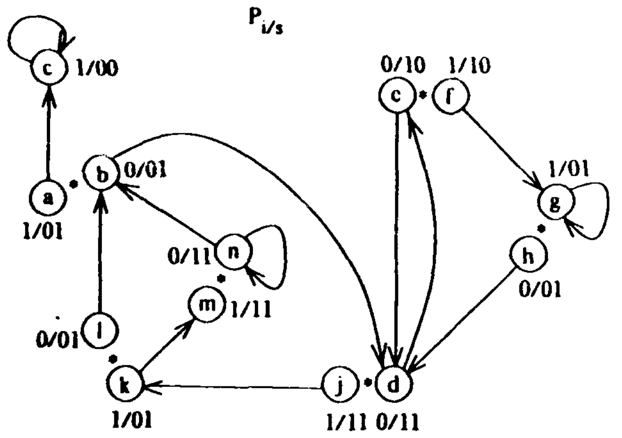


Figure 7-4: Example - Implementation program

Relation $F_{s/i}$:

$\{\langle a,c\rangle, \langle b,d\rangle, \langle c,c\rangle, \langle d,e\rangle, \langle e,i\rangle, \langle f,g\rangle, \langle g,g\rangle, \langle h,i\rangle,$
$\langle j,k\rangle, \langle k,m\rangle, \langle l,b\rangle, \langle m,m\rangle, \langle n,b\rangle\}$

Relation $F_{i/s}$:

$\{\langle a,c\rangle, \langle b,d\rangle, \langle c,c\rangle, \langle d,e\rangle, \langle e,d\rangle, \langle f,g\rangle, \langle g,g\rangle, \langle h,d\rangle,$
$\langle j,k\rangle, \langle k,m\rangle, \langle l,b\rangle, \langle m,m\rangle, \langle n,b\rangle\}$

Relation R_{si}:

$\{\langle a,a\rangle, \langle b,b\rangle, \langle c,c\rangle, \langle d,d\rangle, \langle e,e\rangle, \langle f,f\rangle, \langle g,g\rangle, \langle h,h\rangle,$
$\langle i,d\rangle, \langle j,j\rangle, \langle k,k\rangle, \langle l,l\rangle, \langle m,m\rangle, \langle n,n\rangle\}$

Figure 7-5: Example - Relations defining model

Relation $R_{si}F_{i/s}$:

$\{\langle a,c\rangle, \langle b,d\rangle, \langle c,c\rangle, \langle d,e\rangle, \langle e,d\rangle, \langle f,g\rangle, \langle g,g\rangle, \langle h,d\rangle,$
$\langle i,e\rangle, \langle j,k\rangle, \langle k,m\rangle, \langle l,b\rangle, \langle m,m\rangle, \langle n,b\rangle\}$

Relation $F_{s/i}R_{si}$:

$\{\langle a,c\rangle, \langle b,d\rangle, \langle c,c\rangle, \langle d,e\rangle, \langle e,d\rangle, \langle f,g\rangle, \langle g,g\rangle, \langle h,d\rangle,$
$\langle j,k\rangle, \langle k,m\rangle, \langle l,b\rangle, \langle m,m\rangle, \langle n,b\rangle\}$

Relation $R_{is}F_{s/i}$:

$\{\langle a,c\rangle, \langle b,d\rangle, \langle c,c\rangle, \langle d,e\rangle, \langle e,i\rangle, \langle f,g\rangle, \langle g,g\rangle, \langle h,i\rangle,$
$\langle j,k\rangle, \langle k,m\rangle, \langle l,b\rangle, \langle m,m\rangle, \langle n,b\rangle\}$

Relation $F_{i/s}R_{is}$:

$\{\langle a,c\rangle, \langle b,d\rangle, \langle b,i\rangle, \langle c,c\rangle, \langle d,e\rangle, \langle e,d\rangle, \langle e,i\rangle, \langle f,g\rangle,$
$\langle g,g\rangle, \langle h,d\rangle, \langle h,i\rangle, \langle j,k\rangle, \langle k,m\rangle, \langle l,b\rangle, \langle m,m\rangle, \langle n,b\rangle\}$

Figure 7-6: Four relations for test for consistency

consideration is that any action taken by the implementation should correspond either to an action declared in the specification or to a don't-care condition in the specification. Eq. (7.7) clearly satisfies this requirement, but Eq. (7.8) is also acceptable.

Assume we have deterministic programs $P_{s/i}$ and $P_{i/s}$, and relation R_{is} satisfying Eq. (7.8). Suppose some action $u\ F_{i/s}\ v$ performed by $P_{i/s}$ corresponds to neither an edge nor a don't-care condition in the specification. Figure 7-7 shows the problem. Vertex u is related to some vertex s by R_{is}. Since u is not related to a don't-care vertex, there exists an edge $\langle s,t\rangle$ in $F_{s/i}$. Since $\langle u,v\rangle$ does not correspond to $\langle s,t\rangle$, we have $v \notin R_{si}(t)$. ($R_{si}(t)$ is the set of states w such that $t\ R_{si}\ w$.)

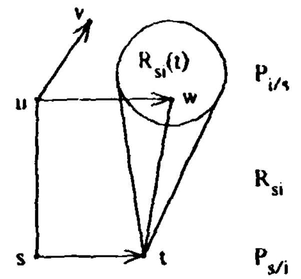


Figure 7-7: Incorrect programs satisfying Eq. (7.8)

We have $u\ R_{is}\ s\ F_{s/i}\ t$. Therefore, from Eq. (7.8) we also have $u\ F_{i/s}\ w\ R_{is}\ t$ for some state w in $R_{si}(t)$. Now $P_{i/s}$ specifies both $u\ F_{i/s}\ w$ and $u\ F_{i/s}\ v$, so $P_{i/s}$ is nondeterministic. This contradicts the condition that the programs are deterministic. Therefore, Eq. (7.8) guarantees the third desirable property for deterministic programs with don't-cares.

7.7.3. Nondeterministic Programs without Don't-Cares

Again, three issues arise in defining consistency for nondeterministic programs without don't-cares.

1. For deterministic models, it is desirable that the specification be completely implemented. For nondeterministic models, this is not necessary. Suppose a specification indicates three possible successor states for state s. An implementation which always chooses the same one of those three states would be correct, even though it does not completely implement the specification. We do not require that each possible successor is sometimes the actual successor. We care only that the actual successor is one of the specified possible successors.

If for some case we do wish to require that each possible successor is sometimes the actual successor, then we can introduce a dummy input variable. We then choose the successor deterministically from the input value. Brand made this point in his technical report [Br78]. Therefore, the *complete* implementation of the specification is not an issue for nondeterministic models.

2. Even though the complete implementation of all edges in the specification $F_{s/i}$ is not necessary, we should implement at least one edge incident out of each vertex in $D_{s/i}$. Eq. (7.8) obviously guarantees this by requiring the implementation of every edge. However, Eq. (7.7) is also satisfactory.

Consider arbitrary vertex s in $D_{s/i}$. It is related to some vertex u in $D_{i/s}$ by R_{si}. *Since there are no don't-cares, $u\ F_{i/s}\ v$ for some vertex v. Then $s\ R_{si} F_{i/s}\ v$. If* Eq. (7.7) *is satisfied, then $s\ F_{s/i}\ t\ R_{si}\ v$ for some vertex t. Therefore, one edge* incident out of s is implemented.

3. When don't-cares were allowed, we did not require every edge in $F_{i/s}$ to be explicitly mirrored in $F_{s/i}$. However, when don't-cares are not allowed, this requirement is necessary. Eq. (7.7) enforces this requirement, while Eq. (7.8) fails to do so, as noted by Brand [Br78].

The first two points above do not discriminate between Eqs. (7.7) and (7.8). The third point motivates the preference for Eq. (7.7), the implied specification, for nondeterministic program models without don't-cares.

7.7.4. Nondeterministic Programs with Don't-Cares

A nondeterministic program with don't-cares can be converted to a nondeterministic program without don't-cares. This is done by specifying every vertex in $D_{a/b}$ as a possible successor to every don't-care vertex in $D_{a/b}$. The discussion for nondeterministic programs without don't-cares then applies, and Eq. (7.7) is preferred.

The addition of these edges can drastically increase the size of the edge set. We can replace the test of Eq. (7.7) against the programs without don't-care vertices by an equivalent test against the program with don't-cares.

Let $P_{s/i} = \langle D_{s/i}, F_{s/i} \rangle$ and $P_{i/s} = \langle D_{i/s}, F_{i/s} \rangle$ be nondeterministic programs with don't-care vertices. Also let $D^x_{s/i}$ and $D^x_{i/s}$ be the don't-care vertices in $P_{s/i}$ and $P_{i/s}$. Then the application of Eq. (7.7) to the programs with don't-care edges added is described by

$$R_{si}(F_{i/s} \cup (D^x_{i/s} \times D_{i/s})) \subseteq (F_{s/i} \cup (D^x_{s/i} \times D_{s/i}))R_{si} \quad . \qquad (7.12)$$

We will assume that no one vertex s in $D_{s/i}$ has immediate successors $F_{s/i}(\{s\})$ related by R_{si} to all vertices in $D_{i/s}$. This is expressed by

$$\text{not } (\exists s \in D_{s/i} \mid R_{si}(F_{s/i}(\{s\})) = D_{i/s}) \quad . \qquad (7.13)$$

Note that this condition applies *before* the addition of don't-care edges. Under this condition, Eq. (7.12) is equivalent to the following tests on $P_{s/i}$, $P_{i/s}$, and R_{si}:

$$R_{si}F_{i/s} \subseteq (F_{s/i}R_{si}) \cup (D^x_{s/i} \times D_{i/s}) \quad . \qquad (7.14)$$

$$R_{is}(D^x_{i/s}) \subseteq D^x_{s/i} \quad . \qquad (7.15)$$

Eqs. (7.14) and (7.15) are preferable to Eq. (7.12) because the left hand side of Eq. (7.14) is generally much smaller than the left hand side of Eq. (7.12). If Eq. (7.13) does not hold, then Eqs. (7.14) and (7.15) are stronger than Eq. (7.12); under no condition are they weaker.

7.8. Deterministic Model Reconsidered

Section 7.7.2 shows that Eq. (7.8) is appropriate for deterministic models with don't-cares. Sections 7.7.3 and 7.7.4 show that Eq. (7.7) is better for nondeterministic models, with or without don't-cares. This seems paradoxical, *because a deterministic model is just a special case of a nondeterministic model*, and the same type of test should be suitable for both.

To resolve the paradox, note that in Section 7.7.4, we had to modify either the programs, adding don't-care edges, or the consistency criterion, using Eqs. (7.14) and (7.15) instead of (7.7). We may conjecture that the application of Eqs. (7.14) and (7.15) is equivalent to the application of Eq. (7.8) for deterministic models with don't-cares. This section proves such a conjecture.

The following lemmas and theorem assume we are given programs P_s and P_i and relation $R_{si} \subseteq D_s \times D_i$. Assume that F_s and F_i are deterministic with don't-cares allowed. That is, each vertex in D_s and D_i has exactly zero or one successor. Define $P_{s/i}$ and $P_{i/s}$ by Eq. (7.6). Finally, $D^x_{s/i} \subseteq D_{s/i}$ and $D^x_{i/s} \subseteq D_{i/s}$ are the sets of vertices that have no successors in $P_{s/i}$ and $P_{i/s}$; the don't-care vertices.

This section will show that Eqs. (7.14) and (7.15) are equivalent to Eq. (7.8) for these deterministic models. The equations are repeated here:

$$R_{si} F_{i/s} \subseteq (F_{s/i} R_{si}) \cup (D^x_{s/i} \times D_{i/s}) \quad . \tag{7.14}$$

$$R_{is}(D^x_{i/s}) \subseteq D^x_{s/i} \quad . \tag{7.15}$$

$$R_{is} F_{s/i} \subseteq F_{i/s} R_{is} \quad . \tag{7.8}$$

Lemma 7.1: If Eqs. (7.14) and (7.15) hold for deterministic $P_{s/i}$, $P_{i/s}$, and R_{si}, then Eq. (7.8) also holds. □

PROOF: Consider arbitrary vertices u, s, and t such that $u\ R_{is}\ s\ F_{s/i}\ t$. It will suffice to show that $u\ F_{i/s} R_{is}\ t$. *Suppose that* $u \in D^x_{i/s}$. *Then by Eq. (7.15),* $s \in D^x_{s/i}$. This contradicts the fact that s has a successor (t), so we must have instead $u \notin D^x_{i/s}$. Then $v = F_{i/s}(u)$ is defined.

Now $s\ R_{si}\ u\ F_{i/s}\ v$, so from Eq. (7.14), $s\ F_{s/i} R_{si}\ v$, since $s \notin D^x_{s/i}$. Since $F_{s/i}$ is deterministic, then we necessarily have $s\ F_{s/i}\ t\ R_{si}\ v$. Then $u\ F_{i/s}\ v\ R_{is}\ t$. **QED.**

Lemma 7.2: If Eq. (7.8) holds for deterministic $P_{s/i}$, $P_{i/s}$, and R_{si}, then Eq. (7.14) also holds. □

PROOF: Consider arbitrary vertices s, u, and v such that $s\ R_{si}\ u\ F_{i/s}\ v$. It will suffice to show that if $s \notin D^x_{s/i}$, then $s\ F_{s/i} R_{si}\ v$. Assume $s \notin D^x_{s/i}$. Then $t = F_{s/i}(s)$ is defined.

Now $u\ R_{is}\ s\ F_{s/i}\ t$, so from Eq. (7.8), $u\ F_{i/s} R_{is}\ t$. Since $F_{i/s}$ is deterministic, then we necessarily have $u\ F_{i/s}\ v\ R_{is}\ t$. Then $s\ F_{s/i}\ t\ R_{si}\ v$. **QED.**

Lemma 7.3: If Eq. (7.8) holds for deterministic $P_{s/i}$, $P_{i/s}$, and R_{si}, then Eq. (7.15) also holds. □

PROOF: Consider arbitrary vertices u and s such that $u\ R_{is}\ s$ and $u \in D^x_{i/s}$. We must show that $s \in D^x_{s/i}$. Suppose this is not the case. Then $t = F_{s/i}(s)$ is defined, and we have $u\ R_{is}\ s\ F_{s/i}\ t$. Eq. (7.8) then implies that $u\ F_{i/s} R_{is}\ t$. This contradicts the original condition that u has no successors. **QED.**

Theorem 7.4: Eq. (7.8) holds for deterministic $P_{s/i}$, $P_{i/s}$, and R_{si} if and only if Eqs. (7.14) and (7.15) hold. □

PROOF: This follows directly from Lemmas 7.1, 7.2, and 7.3. **QED.**

This theorem unifies the consistency criterion for the several program models considered. Note that if $P_{s/i}$ and $P_{i/s}$ have no don't-cares, then Eqs. (7.14) and (7.15) reduce to Eq. (7.7). Then Eqs. (7.14) and (7.15) constitute a criterion for consistency that is suitable for all four program models discussed in Section 7.7:

- deterministic models without don't-cares (reduce to Eq. (7.7) in Section 7.7.1),
- deterministic models with don't-cares (this section),
- nondeterministic models without don't-cares (reduce to Eq. (7.7) in Section 7.7.3), and
- nondeterministic models with don't-cares (Section 7.7.4).

We will call Eqs. (7.14) and (7.15) the "simple" criterion.

7.9. Closure Again

Section 7.4 indicated that Brand had a practical reason for identifying states in the mapping R_{si} solely by their location counters. Here, we discuss this reason and suggest a more practical criterion for consistency that allows the use of mappings defined in this way. This definition will be related to the simpler definition from the preceding development.

7.9.1. Stopping Points in Loops

Consider the programs $P_s = \langle D_s, F_s \rangle$ and $P_i = \langle D_i, F_i \rangle$ and relation $R_{si} \subseteq D_s \times D_i$ (which defines new programs $P_{s/i}$ and $P_{i/s}$). Also define sets $D^x_{s/i} \subseteq D_{s/i}$ and $D^x_{i/s} \subseteq D_{i/s}$ containing vertices with no successors in $P_{s/i}$ and $P_{i/s}$. The criterion for consistency of P_i with P_s is that P_s, P_i, and R_{si} satisfy Eqs. (7.14) and (7.15).

The program models P_s and P_i are so complex that this operation carried out for individual state pairs is intractable. However, symbolic simulation allows the analysis of many state pairs simultaneously. The states in $D_{a/b}$ are sometimes called "stopping points" [Bi78], because in a symbolic simulation carrying out the above test, the simulation stops when it reaches a set of states in $D_{a/b}$.

A symbolic simulation including a loop that does not contain any stopping points can generate many cases, one case for each possible number of passes through the loop. For this reason, it is advantageous to include in $D_{a/b}$ a state from each pass through a loop in P_a. An easy way to do this is to identify stopping points solely by their location counters, and then to specify at least one such stopping point

inside each loop. This is the practical consideration behind Brand's argument in Section 7.4.

7.9.2. New Criterion for Consistency

Figure 7-8a repeats Figure 7-2, in which a simple operation in P_i implements a loop in P_s. This figure does not satisfy Eq. (7.14), because $s_i\ R_{si}\ u\ F_{i/s}\ v$, but $s_i \notin D^x_{s/i}$, and $\langle s_i, v \rangle \notin F_{s/i} R_{si}$. We argue that nonetheless, P_i and P_s are still consistent: $s_i\ F_{s/i}^*\ t\ R_{si}\ v$ through a sequence of states s_j related to u. A similar remark holds if s_n is a don't-care vertex, as shown in Figure 7-8b.

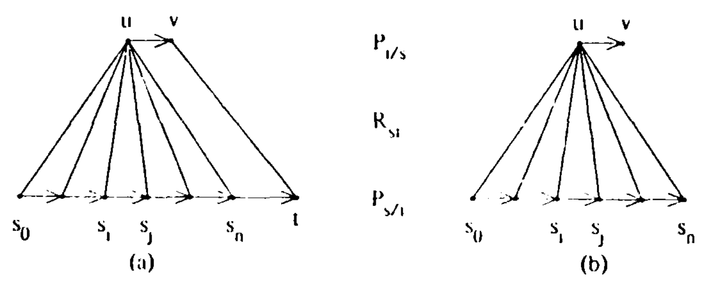


Figure 7-8: Original example motivating use of closure

Now assume that the following condition is satisfied (see Figure 7-9):

$$\forall \text{ states } s, t \in D_{s/i} \text{ and } u, v \in D_{i/s}\text{: } (u\ R_{is}\ s\ F_{s/i}\ t\ R_{si}\ v\ R_{is}\ s) \supset (u\ R_{is}\ t). \quad (7.16)$$

Given $u\ R_{is}\ s_i\ F_{s/i}\ s_j$ with Eq. (7.16) holding, it follows that $s_i\ R_{si}\ u\ R_{is}\ s_j$ if and only if $s_i\ R_{si} R_{is}\ s_j$, as proved here:

PROOF: *Only if:* $s_i\ R_{si} R_{is}\ s_j$ trivially follows from $s_i\ R_{si}\ u\ R_{is}\ s_j$. *If:* Since

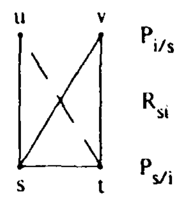


Figure 7-9: Condition for simplified definition

$s_i R_{si} R_{is} s_j$, there exists a state $w \in D_{i/s}$ such that $s_i R_{si} w R_{is} s_j$. Then Eq. (7.16) applies with $\langle s,t,u,v\rangle = \langle s_i,s_j,u,w\rangle$. **QED.**

This extension to the notion of consistency then can be expressed by

$$R_{si}F_{i/s} \subseteq (R_{si}R_{is} \cap F_{s/i})^*((F_{s/i}R_{si}) \cup (D^x_{s/i} \times D_{i/s})) \quad . \qquad (7.17)$$

Now consider the converse situation in Figure 7-10. This figure also does not satisfy Eq. (7.14): $s\, R_{si}\, u_i\, F_{i/s}\, u_j$, but $s \notin D^x_{s/i}$, and $\langle s,u_j\rangle \notin F_{s/i}R_{si}$. Again, we argue that P_i and P_s are consistent. State u_j is actually the same point (same location counter) as u_i, and therefore rightly corresponds not to t, but to s. Eq. (7.18) adds this extension to Eq. (7.14). Relation I is the identity relation.

$$R_{si}F_{i/s} \subseteq ((F_{s/i} \cup I)R_{si}) \cup (D^x_{s/i} \times D_{i/s}) \quad . \qquad (7.18)$$

Eq. (7.18) allows verification in Figure 7-10, but it also incorrectly verifies the example in Figure 7-11. There, P_i specifies that the program stays in a loop, while P_s does not allow staying in a loop. If state u in $D_{i/s}$ is in a loop of states all related to s in $D_{s/i}$, then the successor of s (t in Figure 7-11) should be related by R_{si} to some vertex in the loop. This requirement and Eq. (7.16) imply that the successor of s must be related to every vertex in the loop. Eq. (7.19) tests this condition:

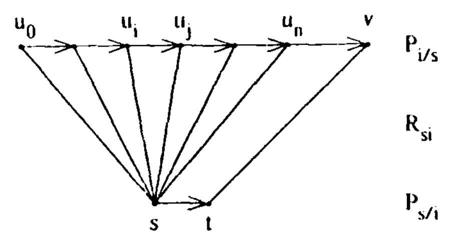


Figure 7-10: Second example motivating use of closure

$$R_{si}((R_{is}R_{si} \cap F_{i/s})+ \cap I) \subseteq (F_{s/i}R_{si}) \cup (D^x_{s/i} \times D_{i/s}) \quad . \qquad (7.19)$$

The plus sign denotes +-closure; if Q is a relation, then $Q+ = QQ^*$.

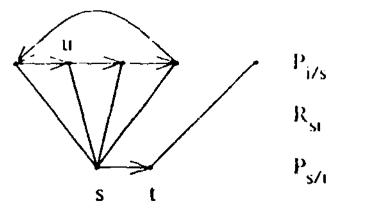


Figure 7-11: Incorrect verification by Eq. (7.18)

The combination of Eqs. (7.17) and (7.18) yields Eq. (7.20):

$$R_{si}F_{i/s} \subseteq (R_{si}R_{is} \cap F_{s/i})^*(((F_{s/i} \cup I)\, R_{si}) \cup (D^x_{s/i} \times D_{i/s})) \quad . \qquad (7.20)$$

Eq. (7.20) puts closure back into Eq. (7.14), but in a limited fashion that retains intuitive notions of correctness. Eqs. (7.15), (7.19), and (7.20) comprise a criterion for consistency that supports stopping points in loops in both P_s and P_i, subject to the condition in Eq. (7.16). We will call this criterion the "extended" criterion.

7.10. Relation between Extended and Simple Criteria

7.10.1. . review

The object of Section 7.10 is to demonstrate that the extended criterion for consistency, designed to facilitate the verification of programs with loops, still retains intuitive notions of correctness. The approach is to show that

- if P_i is consistent with P_s under R_{si} by the extended criterion from Section 7.9,
- then P_i is also consistent with P_s under a reasonable simulation relation R'_{si} by the simple criterion from Section 7.8.

Therefore, the extended criterion retains the intuitive notions of correctness captured by the simple definition. This will be proven only for deterministic models.

This discussion takes place in three parts.

First, Section 7.10.2 defines the new relation R'_{si}. To do this, it defines an equivalence relation on $D_{a/b}$ and selects a distinguished ("terminal") vertex from each equivalence class. Relation R'_{si} is then defined on the distinguished vertices.

Second, Section 7.10.3 demonstrates that R'_{si} is a reasonable simulation relation. It does this by showing that "important" behavioral details captured by R_{si} are also captured by R'_{si}.

Third, Section 7.10.4 shows that P_i is consistent with P_s under R'_{si} by the simple criterion, assuming consistency under R_{si} by the extended criterion.

The discussion in Section 7.10 will assume that the program model is deterministic with don't-cares allowed. For each state $s \in D_a$, there is exactly zero or one state t such that $s \; F_a \; t$. Note that $P_{a/b}$ will also be deterministic with don't-cares.

Given programs P_s and P_i and relation $R_{si} \subseteq D_s \times D_i$ which satisfy Eqs. (7.15), (7.19), and (7.20), we will show how to derive relation R'_{si} satisfying Eqs. (7.14) and (7.15). This discussion makes the following assumptions about P_s, P_i, R_{si}, programs $P_{s/i}$ and $P_{i/s}$ defined by Eq. (7.6), and sets $D^x_{s/i}$ and $D^x_{i/s}$ defined by Eq. (7.5).

- All programs P_a have a finite number of states in D_a.
- In all programs P_a, F_a is a function with domain a subset of D_a:

$$\forall \text{ vertices } s, t, r \text{ in } D_a\text{:} \; (s \; F_a \; t \text{ and } s \; F_a \; r) \supset (t = r) \,. \qquad (7.21)$$

- Eqs. (7.15), (7.19), and (7.20) are satisfied:

$$R_{is}(D^x_{i/s}) \subseteq D^x_{s/i} \,. \qquad (7.15)$$

$$R_{si}((R_{is} R_{si} \cap F_{i/s})+ \; \cap \; I) \subseteq (F_{s/i} R_{si}) \cup (D^x_{s/i} \times D_{i/s}) \,. \qquad (7.19)$$

$$R_{si} F_{i/s} \subseteq (R_{si} R_{is} \cap F_{s/i})^* (((F_{s/i} \cup I) R_{si}) \cup (D^x_{s/i} \times D_{i/s})) \,. \qquad (7.20)$$

- The following holds. This includes Eq. (7.16) and the symmetric condition.

$\forall$ states $s, t \in D_{a/b}$ and $u, v \in D_{b/a}$:

$$(s \; R_{ab} \; u \; F_{b/a} \; v \; R_{ba} \; t \; R_{ab} \; u) \supset (s \; R_{ab} \; v) \,. \qquad (7.22)$$

7.10.2. New Relation

Define relation R_{aa} on $D_{a/b} \times D_{a/b}$:

$$R_{aa} = (R_{ab} R_{ba} \cap (F_{a/b} \cup F_{a/b}^{-1}))^* \quad . \tag{7.23}$$

Relation R_{aa} is an equivalence relation since it is reflexive, symmetric, and transitive. Denote by $R_{aa}[s]$ the equivalence class defined by R_{aa} that contains vertex s. Relations R_{ss} and R_{ii} are defined by Eq. (7.23), with resulting equivalence classes $R_{ss}[s]$ and $R_{ii}[u]$.

Now define a *terminal vertex* in $D_{a/b}$ to be any vertex t reachable from all other vertices in $R_{aa}[t]$ via a path in $R_{aa}[t]$:

t is a terminal vertex *iff*

$$\forall \text{ vertices } s \in R_{aa}[t]: \; s \, (R_{ab} R_{ba} \cap F_{a/b})^* \, t \quad . \tag{7.24}$$

Choose a set $D'_{a/b}$, a set of terminal vertices with exactly one terminal vertex from each equivalence class. If an equivalence class has more than one terminal vertex, the choice of the terminal vertex is arbitrary. (Lemma 7.5 will show that every equivalence class has at least one terminal vertex.) Define $t_{ae}[s]$ to be the terminal vertex of $R_{aa}[s]$ that is in $D'_{a/b}$. (Subscript "e" denotes "equivalence class.")

Terminal vertices in $D_{s/i}$ and $D_{i/s}$ and the vertex sets $D'_{s/i}$ and $D'_{i/s}$ are defined in this way. Then define the new relation R'_{si} by

$$R'_{si} = R_{si} \cap (D'_{s/i} \times D'_{i/s}) \quad . \tag{7.25}$$

New *programs* $P'_{s/i} = \langle D'_{s/i}, F'_{s/i} \rangle$ and $P'_{i/s} = \langle D'_{i/s}, F'_{i/s} \rangle$ are then defined from P_s, P_i, and R'_{si} by Eq. (7.6). Don't-care vertex sets $D'^{x}_{s/i}$ and $D'^{x}_{i/s}$ are defined for $P'_{s/i}$ and $P'_{i/s}$ by Eq. (7.5). The major claims of this discussion will be that R'_{si} is a suitable relation, and that P_s, P_i, and R'_{si} satisfy Eqs. (7.14) and (7.15):

$$R'_{si} F'_{i/s} \subseteq (F'_{s/i} R'_{si}) \cup (D'^{x}_{s/i} \times D'_{i/s})$$
$$R'_{is}(D'^{x}_{i/s}) \subseteq D'^{x}_{s/i} \quad .$$

7.10.3. Suitability of New Relation

Note that it is trivial to define a relation R''_{si} that satisfies Eqs. (7.14) and (7.15) given arbitrary programs P_s and P_i. In fact, the empty set is such a relation. Obviously, the empty set does not capture the desired intuitive correspondence between P_s and P_i.

However, R'_{si} does capture this correspondence as expressed by R_{si}, in the following sense: For all vertices u and v in $D_{a/b}$ but not in the same equivalence class defined by R_{aa}, we have (Figure 7-12)

- if $u \; F_{a/b} \; v$, then $u \; F'_{a/b} \; t_{ae}[v]$; also, $u \; R'_{ab} \; s$ for some vertex s in $D'_{b/a}$ (this guarantees that $u \; F_{i/s} \; v$ is a correct edge if Eqs. (7.14) and (7.15) is satisfied with R'_{si}); and
- if $u \; F'_{a/b} \; v$, then there is a vertex w such that $u \; F_{a/b} \; w$ and $v = t_{ae}[w]$.

Figure 7-12 illustrates these assertions. In the figure, boxed vertices are in $D'_{a/b}$ or $D'_{b/a}$, and dashed edges are edges whose existence is implied by the presence of other solid edges. A small oval denotes an equivalence class, and a wavy edge denotes a path within a single equivalence class.

Proofs of the above assertions follow. Each theorem directly proves a portion of the assertions. The lemmas facilitate the proof of the theorems. **All lemmas and theorems below assume the definitions and assumptions from Sections 7.10.1 and 7.10.2, including Eqs. (7.21) through (7.25).** The reader may wish to skip the proofs on first reading.

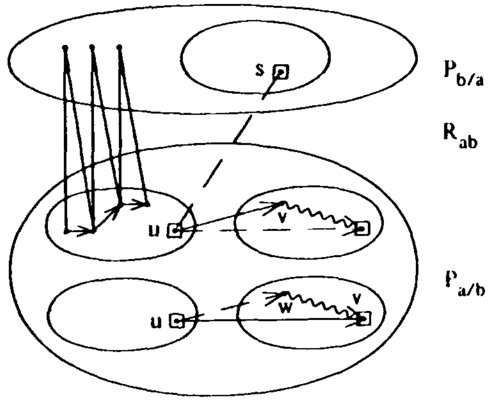


Figure 7-12: Properties of new simulation relation

Lemma 7.5: Every equivalence class $R_{aa}[s]$ has at least one terminal vertex. □

PROOF: Select an arbitrary vertex s in $D_{a/b}$ and look at its descendants in the directed edge set $(R_{ab}R_{ba} \cap F_{a/b})$. Since $D_{a/b}$ has a finite number of states, one of three cases in Figure 7-13 arises: (a) The path reaches a vertex that has no successor. (b) The path ends at a self-loop. (c) The path enters a larger loop.

Consider case (a). Let t be the vertex with no successor. For every vertex m in $R_{aa}[t]$, there exists a loop-free path $m\ (R_{ab}R_{ba} \cap (F_{a/b} \cup F_{b/a}^{-1}))^*\ t$, as illustrated in Figure 7-14. For each pair of adjacent vertices j and k in this path, either $j\ F_{a/b}\ k$ or $k\ F_{a/b}\ j$. In particular, in Figure 7-14, either $p\ F_{a/b}\ t$ or $t\ F_{a/b}\ p$. But t has no successor in the edge set $(R_{ab}R_{ba} \cap F_{a/b})$. Therefore $p\ F_{a/b}\ t$.

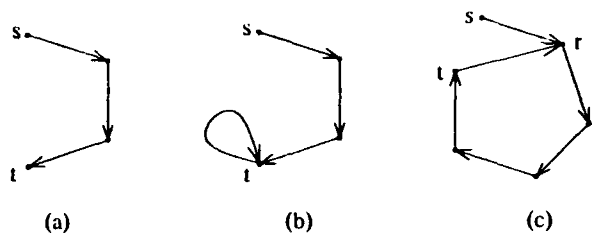


Figure 7-13: Three types of terminal vertices

Moving to the next pair, either $p\ F_{a/b}\ q$ or $q\ F_{a/b}\ p$. But p has at most one successor in the edge set $(R_{ab}R_{ba} \cap F_{a/b})$ (Eq. (7.21)), $p\ F_{a/b}\ t$, and t is distinct from q (since the path from m to t is loop-free). Therefore $q\ F_{a/b}\ p$. Applying this step inductively, we find $m\ (R_{ab}R_{ba} \cap F_{a/b})^*\ t$. This satisfies Eq. (7.24), so t is by definition a terminal vertex.

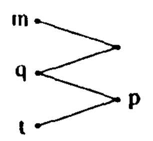


Figure 7-14: Reachability of terminal vertex for case (a)

In cases (b) and (c) of Figure 7-13, we can remove edge $\langle t,t \rangle$ or $\langle t,r \rangle$ without modifying the equivalence class partition. The edge removal changes these cases to case (a), since t then has no successor. In each modified case, t is a terminal vertex by the above analysis. Putting the deleted edge back into $F_{a/b}$ does not alter the reachability of t, so t is a terminal vertex in each original case as well. **QED.**

Lemma 7.6 says that if two vertices are related by R_{si}, then all their successors in the same equivalence classes are also related by R_{si}.

Lemma 7.6: Given arbitrary vertices s in $D_{s/i}$ and u in $D_{i/s}$: If $s\ R_{si}\ u$, then all vertices reachable from s by $(R_{si}R_{is} \cap F_{s/i})^*$ (including s) are related by R_{si} to all vertices reachable from u by $(R_{is}R_{si} \cap F_{i/s})^*$ (including u). □

PROOF: By induction, it suffices to show that if $s\ (R_{si}R_{is} \cap F_{s/i})\ t$ then $t\ R_{si}\ u$, and if $u\ (R_{is}R_{si} \cap F_{i/s})\ v$ then $s\ R_{si}\ v$. Both of these follow from Eq. (7.22). **QED.**

Lemma 7.7 says that if two vertices are related by R_{si}, then the terminal vertices (in D'_{si} and D'_{is}) of their equivalence classes are also related by R_{si}.

Lemma 7.7: Given arbitrary vertices s in $D_{s/i}$ and u in $D_{i/s}$: If $s\ R_{si}\ u$, then $t_{se}[s]\ R_{si}\ t_{ie}[u]$. □

PROOF: By definition of terminal vertices, $t_{se}[s]$ is reachable from s by $(R_{si}R_{is} \cap F_{s/i})^*$. Similarly, $t_{ie}[u]$ is reachable from u by $(R_{is}R_{si} \cap F_{i/s})^*$. Then Lemma 7.6 applies. **QED.**

Lemma 7.8 says that if any edge $F_{a/b}$ is incident from vertex s to a different equivalence class, then s is the unique terminal vertex in its equivalence class.

Lemma 7.8: Given arbitrary vertices s and t in different equivalence classes $R_{aa}[s]$ and $R_{aa}[t]$: If $s\ F_{a/b}\ t$, then s is the unique terminal vertex for $R_{aa}[s]$. □

PROOF: If s and t are in different equivalence classes and $s\ F_{a/b}\ t$, then vertex s has no successor in the edge set $(R_{ab}R_{ba} \cap F_{a/b})$. This is an example of case (a) in the proof of Lemma 7.5, so s is a terminal vertex. We show it is the only terminal vertex by contradiction. Assume $R_{aa}[s]$ has a second terminal vertex r distinct from s. By the definition of terminal vertices, r is reachable from s by $(R_{ab}R_{ba} \cap F_{a/b})^*$, and s is reachable from r. Since $F_{a/b}$ is a directed edge set, this implies the existence of a loop including r and s, as shown in Figure 7-15. But then s has two successors, violating Eq. (7.21). Therefore, r cannot be a distinct terminal vertex, so s is unique. **QED.**

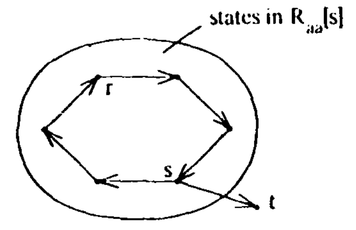


Figure 7-15: Impossibility of two distinct terminal vertices

Lemma 7.9 says that any don't-care vertex in $P_{a/b}$ is the unique terminal vertex in its equivalence class.

Lemma 7.9: Given arbitrary vertex s in $D_{a/b}$: If s is a don't-care vertex in $P_{a/b}$, then s is the unique terminal vertex for $R_{aa}[s]$. □

PROOF: The proof parallels that for Lemma 7.8. Vertex s corresponds to case (a) in the proof of Lemma 7.5, so s is a terminal vertex. If $R_{aa}[s]$ had a second distinct terminal vertex t, then s would have a successor in the edge set $(R_{ab}R_{ba} \cap F_{a/b})$, contradicting the given condition that s is a don't-care vertex in $P_{a/b}$. **QED.**

The following three theorems demonstrate the assertions claimed at the beginning of this section.

Theorem 7.10: Given arbitrary vertices u and v in different equivalence classes $R_{aa}[u]$ and $R_{aa}[v]$: *If* $u\ F_{a/b}\ v$, *then* $u\ F'_{a/b}\ t_{ae}[v]$. □

PROOF: By Lemma 7.8, u is the unique terminal vertex for $R_{aa}[u]$. Therefore u is in $D'_{a/b}$. Vertex $t_{ae}[v]$ is also in $D'_{a/b}$. By definition of a terminal vertex, there exists path(s) $u\ F_{a/b}\ v\ (R_{ab}R_{ba} \cap F_{a/b})^{*}\ t_{ae}[v]$. Choose such a path in which each vertex appears only once (no loops). We have $u\ F_{a}^{*}\ t_{ae}[v]$ along this path. Also, since each equivalence class has only one vertex in $D'_{a/b}$, then only the endpoints of this path are in $D'_{a/b}$. Therefore, by definition of $F'_{a/b}$, $u\ F'_{a/b}\ t_{ae}[v]$. **QED.**

Theorem 7.11: Given arbitrary vertices u and v in different equivalence classes $R_{aa}[u]$ and $R_{aa}[v]$: *If* $u\ F_{a/b}\ v$, *then* $u\ R'_{ab}\ s$ *for some vertex* s *in* $D'_{b/a}$. □

PROOF: Since u is in $D_{a/b}$, there must exist a vertex r in $D_{b/a}$ such that $u\ R_{ab}\ r$. By Lemma 7.7, $t_{ae}[u]\ R_{ab}\ t_{be}[r]$. Since $t_{ae}[u]$ and $t_{be}[r]$ are in $D'_{a/b}$ and $D'_{b/a}$, respectively, then by definition of R'_{ab}, $t_{ae}[u]\ R'_{ab}\ t_{be}[r]$. But by Lemma 7.8, u is the unique terminal vertex for $R_{aa}[u]$, so that $u = t_{ae}[u]$. Then the theorem is true with $s = t_{be}[r]$. **QED.**

Theorem 7.12: Given arbitrary vertices u and v in $D_{a/b}$: *If* $u\ F'_{a/b}\ v$, *then* $u\ F_{a/b}\ w$ *for some vertex* w *such that* $v = t_{ae}[w]$. □

PROOF: If $u\ F'_{a/b}\ v$, then certainly $u\ F_{a/b}\ w$ for some vertex w in $D_{a/b}$. Complete the proof by contradiction: Suppose that $v \neq t_{ae}[w]$. Then $v \notin R_{aa}[w]$ since $v = t_{ae}[v]$. We have $u\ F_{a/b}\ w\ F_{a/b}^{*}\ v$ since $F_{a/b}$ is deterministic. The path $w\ F_{a/b}^{*}\ v$ must exit $R_{aa}[w]$ at some point, and by Lemma 7.8 it can do so only through the unique terminal vertex $t_{ae}[w]$. Therefore we necessarily have

$$u\ F_{a/b}\ w\ F_{a/b}^{*}\ t_{ae}[w]\ F_{a/b} F_{a/b}^{*}\ v \quad .$$

But then a midpoint (namely $t_{ae}[w]$) of any path from u to v is in $D'_{a/b}$, contradicting the given fact that $u\ F'_{a/b}\ v$. Then the supposition that $v \neq t_{ae}[w]$ is false, proving the theorem. **QED.**

Theorems 7.10, 7.11, and 7.12 prove the desirable properties of R'_{si} asserted at the beginning of this section. If $s\ F_{a/b}\ t$, and s and t are in different equivalence classes, then edge $\langle s,t \rangle$ in $F_{a/b}$ will uniquely correspond to edge $\langle s,t_{ae}[t] \rangle$ in $F'_{a/b}$.

If $s\ F_{a/b}\ t$, and s and t are in the same equivalence class, then edge $\langle s,t \rangle$ in $F_{a/b}$ is probably "lost" in $F'_{a/b}$, not corresponding to any edee there. (The exception arises when s and t are both in a loop within $R_{aa}[s]$.) The reasoning is that s and t both are related to the same vertices in $D_{b/a}$ by R_{ab} only to simplify the test for consistency by symbolic simulation; the essential information in R_{ab} is retained if we keep only one stopping point from each equivalence class.

The next section will show that if P_s, P_i, and R_{si} satisfy Eqs. (7.15), (7.19), and (7.20), then P_s, P_i, and R'_{si} will satisfy Eqs. (7.14) and (7.15).

7.10.4. Proof that New Relation Satisfies Simple Criterion

This section will demonstrate that P_i is consistent with P_s under R'_{si} by the simple criterion, assuming consistency under R_{si} by the extended criterion. This is expressed by Theorem 7.13. The following statement of the theorem summarizes the definitions and restrictions assumed in the preceding discussion.

Theorem 7.13: Given the following:

- Programs $P_s = \langle D_s, F_s \rangle$ and $P_i = \langle D_i, F_i \rangle$ and relation $R_{si} \subseteq D_s \times D_i$; D_s and D_i are finite.
- F_s and F_i functions on subsets of D_s and D_i; Eq. (7.21).
- $P_{s/i}$ and $P_{i/s}$ defined from P_s, P_i, and R_{si} by Eq. (7.6).
- $P_{s/i}$, $P_{i/s}$, and R_{si} satisfy Eqs. (7.15), (7.19), (7.20), and (7.22).
- $D'_{s/i}$, subset of $D_{s/i}$ containing exactly one state from each equivalence class generated by relation $(R_{si} R_{is} \cap (F_{s/i} \cup F_{s/i}^{-1}))^*$.
- $D'_{i/s}$, subset of $D_{i/s}$ containing exactly one state from each equivalence class generated by relation $(R_{is} R_{si} \cap (F_{i/s} \cup F_{i/s}^{-1}))^*$.
- $R'_{si} = R_{si} \cap (D'_{s/i} \times D'_{i/s})$.
- $P'_{s/i}$ and $P'_{i/s}$ defined from P_s, P_i, and R'_{si} by Eq. (7.6).
- Don't-care vertex sets $D'^{x}_{s/i}$ and $D'^{x}_{i/s}$ defined from $P'_{s/i}$ and $P'_{i/s}$ by Eq. (7.5).

The following result holds: $P'_{s/i}$, $P'_{i/s}$, and R'_{si} satisfy Eqs. (7.14) and (7.15):

$$R'_{si} F'_{i/s} \subseteq (F'_{s/i} R'_{si}) \cup (D'^{x}_{s/i} \times D'_{i/s}) \quad . \tag{7.26}$$

$$R'_{is}(D'^{x}_{i/s}) \subseteq D'^{x}_{s/i} \quad . \ \square \tag{7.27}$$

First, we state and prove several lemmas. **All the lemmas will assume the conditions given in Theorem 7.13.** Lemmas 7.14 and 7.15 prove useful properties. Lemmas 7.16 and 7.17 prove Eq. (7.26) for two cases, thereby partially proving Theorem 7.13. The theorem will follow directly from the lemmas. The reader may wish to skip the proofs on first reading.

Lemma 7.14 states that if two states in the same equivalence class are related by $F_{a/b}$, then they are also related by $R_{ab} R_{ba}$.

Lemma 7.14: Given states s and t in $D_{a/b}$: If $s \ F_{a/b} \ t$ and $t \in R_{aa}[s]$, then $s \ (R_{ab} R_{ba} \cap F_{a/b}) \ t$. $\square$

PROOF: Prove by contradiction (Figure 7-16). Assume $s \ F_{a/b} \ t$ and $t \in R_{aa}[s]$, but $\langle s,t \rangle \notin (R_{ab} R_{ba} \cap F_{a/b})$. Then $\langle s,t \rangle \notin R_{ab} R_{ba}$. Since s and t are in the same equivalence class, we know s and t are joined by a path $(R_{ab} R_{ba} \cap (F_{a/b} \cup F_{a/b}^{-1}))^*$, which we choose to be loop-free. This path cannot include edge $\langle s,t \rangle$ since $\langle s,t \rangle \notin R_{ab} R_{ba}$. Therefore the situation is as shown in Figure 7-16a. The directed edge is edge $s \ F_{a/b} \ t$. The undirected edges are edges in $(R_{ab} R_{ba} \cap (F_{a/b} \cup F_{a/b}^{-1}))$.

Now $(R_{ab} R_{ba} \cap (F_{a/b} \cup F_{a/b}^{-1})) \subseteq (F_{a/b} \cup F_{a/b}^{-1})$, so for each undirected edge $\langle j,k \rangle$ in the path, either $j \ F_{a/b} \ k$ or $k \ F_{a/b} \ j$. In particular, in Figure 7-16a, either $p \ F_{a/b} \ s$ or $s \ F_{a/b} \ p$. But $s \ F_{a/b} \ t$, $F_{a/b}(s)$ is single-valued, and t is distinct

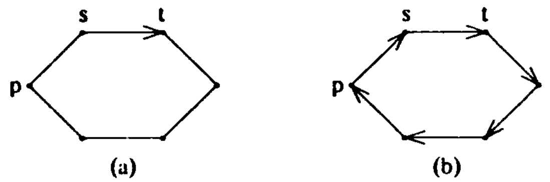


Figure 7-16: Connectivity between states in equivalence class

from p (since $s\ R_{ab}R_{ba}\ p$). Then we cannot have $s\ F_{a/b}\ p$, so $p\ F_{a/b}\ s$. Applying a similar argument along the path recursively, we find $t\ F_{a/b}+\ s$ as shown in Figure 7-16b. Since $j\ R_{ab}R_{ba}\ k$ for each edge $\langle j,k \rangle$ in the path, then $t\ (R_{ab}R_{ba} \cap F_{a/b})+\ s$.

There is a state u in $D_{b/a}$ such that $t\ R_{ab}\ u$. By Lemma 7.6, we also have $s\ R_{ab}\ u$ since s is reachable from t by $(R_{ab}R_{ba} \cap F_{a/b})^*$. (Note that the conditions for Lemma 7.6 are included in the conditions for Theorem 7.13.) Then $s\ R_{ab}\ u\ R_{ba}\ t$, which implies $s\ (R_{ab}R_{ba} \cap F_{a/b})\ t$. This contradicts the original assumption. **QED.**

Now consider an arbitrary vertex s in $D_{a/b}$. Lemma 7.15 says that s is in $D'_{a/b}$ and has no successors in $P'_{a/b}$ if and only if s has no successors in $P_{a/b}$.

Lemma 7.15: $D'^{x}_{a/b} = D^{x}_{a/b}$. □

PROOF: Consider an arbitrary don't care vertex s in $D_{a/b}$ ($s \in D^{x}_{a/b}$). By Lemma 7.9, vertex s is the unique terminal vertex of $R_{aa}[s]$. Then $s \in D'_{a/b}$ and so $s \in D'^{x}_{a/b}$. This shows $D^{x}_{a/b} \subseteq D'^{x}_{a/b}$.

Now suppose some vertex r is in $D'^{x}_{a/b}$ but not $D^{x}_{a/b}$. We have $r\ F_{a/b}\ t$ for some vertex $t \in D_{a/b}$. Then $r\ F_{a/b}+\ t_{ae}[t]$, so $r \notin D'^{x}_{a/b}$, since $t_{ae}[t]$ is in $D'_{a/b}$. This contradicts the supposition that $r \in D'^{x}_{a/b}$. This shows $D'^{x}_{a/b} \subseteq D^{x}_{a/b}$. **QED.**

Lemmas 7.16 and 7.17 prove Eq. (7.26) in Theorem 7.13. Each lemma assumes a vertex pair $\langle s,v \rangle$ that is in the left hand side (LHS) of Eq. (7.26). This implies the existence of a vertex u also assumed in the lemmas. Either u and v are in the same equivalence class or they are not; the lemmas cover each case in turn.

Now if s is in $D'^{x}_{s/i}$, then Eq. (7.26) is immediately satisfied. The lemmas therefore assume that s is not in $D'^{x}_{s/i}$ and show that Eq. (7.26) is still satisfied.

Lemma 7.16 covers the case where u and v are in the same equivalence class, illustrated in Figure 7-17. To avoid clutter, arrows are shown for only one edge in each loop in the figure.

Lemma 7.16: Given states s in $D'_{s/i}$ and u and v in $D'_{i/s}$: If $s\ R'_{si}\ u\ F'_{i/s}\ v$, $s \notin D'^{x}_{s/i}$, and v is in $R_{ii}[u]$, then $s\ F'_{s/i}R'_{si}\ v$. □

PROOF: Make the following observations. (In proofs with numbered paragraphs, the text in each paragraph justifies the initial assertion in the paragraph.)

(1) $s\ R'_{si}\ v\ F'_{i/s}\ v$. Since each equivalence class in $D_{i/s}$ has only one state in $D'_{i/s}$, then $u = v$.

(2) $v\ F_{i/s}+\ v$ through a sequence of vertices in $R_{ii}[v]$. From (1) by definition of $F'_{i/s}$, $v\ F_{i/s}+\ v$ along a path that does not include any other states in

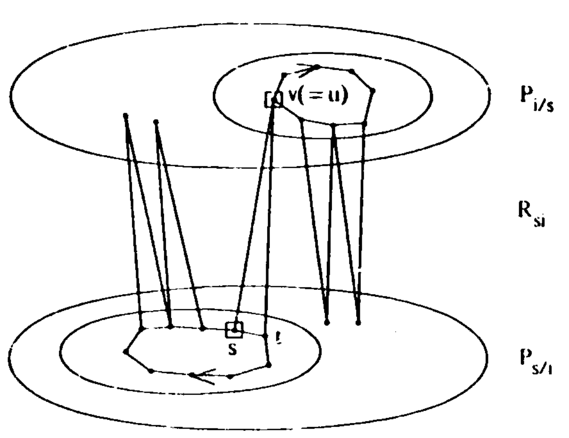


Figure 7-17: Proof of Eq. (7.26): first case

$D'_{i/s}$. But if the path leaves $R_{ii}[v]$, it must necessarily pass through another state in $D'_{i/s}$ when it returns to $R_{ii}[v]$ (see Lemma 7.8 and proof of Theorem 7.12).

(3) $v\ (R_{is}R_{si} \cap F_{i/s})+\ v$, by recursive application of Lemma 7.14 to (2).

(4) $s\ R_{si}\ v$, from (1) since R'_{si} is a subset of R_{si}.

(5) $s\ F_{s/i}\ t\ R_{si}\ v$ for some state t in $D_{s/i}$, from (3) and (4) by Eq. (7.19); note $s \notin D'^{x}_{s/i}$, and $D'^{x}_{s/i} = D^{x}_{s/i}$ by Lemma 7.15.

(6) $s\ (R_{si}R_{is} \cap F_{s/i})\ t$, from (4) and (5).

(7) $t\ (R_{si}R_{is} \cap F_{s/i})^{*}\ s$, because $s \in D'_{s/i}$ is a terminal vertex, and from (6), t is in $R_{ss}[s]$.

(8) $s\ (R_{si}R_{is} \cap F_{s/i})+\ s$, from (6) and (7).

(9) $s\ F_{s/i}+\ s$ through a path in $R_{ss}[s]$, from (8).

(10) $s\ F'_{s/i}\ s$, from (9), since s is the only state from $R_{ss}[s]$ in $D'_{s/i}$.

(11) $s\ F'_{s/i}R'_{si}\ v$, from (1) and (10). **QED.**

Lemma 7.17 covers the case where u and v are in different equivalence classes, illustrated in Figure 7-18. The figure shows s separate from t to facilitate following the proof, although the proof will show $s = t$.

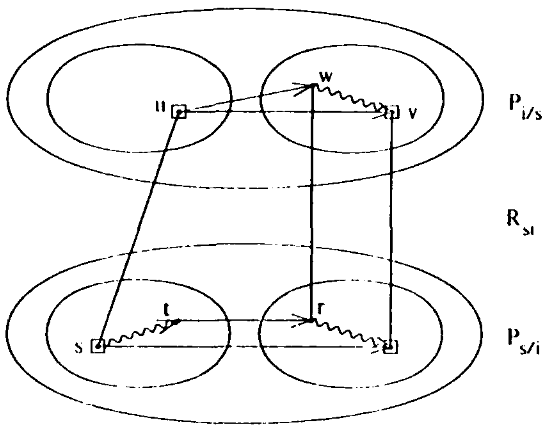


Figure 7-18: Proof of Eq. (7.26): second case

Lemma 7.17: Given states s in $D'_{s/i}$ and u and v in $D'_{i/s}$: If $s\ R'_{si}\ u\ F'_{i/s}\ v$, $s \notin D'^{x}_{s/i}$, and v is not in $R_{ii}[u]$, then $s\ F'_{s/i}R'_{si}\ v$. □

PROOF: Make the following observations.

(1) $s\ R'_{si}\ u\ F'_{i/s}\ v$, with v not in $R_{ii}[u]$, by assumption.

(2) $u\ F_{i/s}\ w\ (R_{is}R_{si} \cap F_{i/s})^{*}\ v$ for some state w in $D_{i/s}$, from (1) and Theorem 7.12, and by definition of a terminal vertex. Note that w is in the same equivalence class as v, and therefore in a different equivalence class from u.

(3) $s\ R_{si} F_{i/s}\ w$, from (1) and (2), since $R'_{si} \subseteq R_{si}$.

(4) $\langle s,w \rangle \notin R_{si}$; otherwise, we would have $u\ (R_{is} R_{si} \cap F_{i/s})\ w$. This would imply $w \in R_{ii}[u]$, contradicting (2).

(5) $s\ (R_{si} R_{is} \cap F_{s/i})^*\ t\ (((F_{s/i} \cup I) R_{si}) \cup (D^x_{s/i} \times D_{i/s}))\ w$ for some state t, by (3) and Eq. (7.20).

(6) $t \in D^x_{s/i}$ by assumption. (This will lead to a contradiction.)

(7) $s = t$. From (1), $s = t_{se}[s]$. Step (5) gives $t_{se}[s] = t_{se}[t]$. Finally, by Lemma 7.9, (6) implies $t_{se}[t] = t$.

(8) $s \in D^x_{s/i}$, from (6) and (7). This contradicts the given condition that $s \notin D^x_{s/i}$.

(9) $t \notin D^x_{s/i}$. *The assumption of* $t \in D^x_{s/i}$ *in* (6) led to a contradiction in (8).

(10) $s\ (R_{si} R_{is} \cap F_{s/i})^*\ t\ (F_{s/i} \cup I)\ r\ R_{si}\ w$ for some vertex r, from (5) and (9).

(11) $s \neq r$, because $r\ R_{si}\ w$ by (10), but $\langle s,w \rangle \notin R_{si}$ by (4).

(12) WLOG assume $s\ (R_{si} R_{is} \cap F_{s/i})^*\ t\ F_{s/i}\ r\ R_{si}\ w$. If $t\ I\ r$ in (10), then from (11), $s \neq t$, so $s\ (R_{si} R_{is} \cap F_{s/i})^+\ t$. Then we can redefine t to denote the predecessor of r in this path.

(13) $r \in R_{ss}[t]$ by assumption. (This will lead to a contradiction.)

(14) $s\ R_{si}\ w$. We have $r\ R_{si}\ w$ from (12), and $s = t_{se}[r]$ from (12) and (13), since $s = t_{se}[s]$ by (1). Therefore s is reachable from r by $(R_{si} R_{is} \cap F_{s/i})^*$. Then Lemma 7.6 gives $s\ R_{si}\ w$. This directly contradicts (4).

(15) r and t are in different equivalence classes. The assumption of $r \in R_{ss}[t]$ in (13) led to a contradiction in (14).

(16) $s = t$. From (1), $s = t_{se}[s]$. Step (12) gives $t_{se}[s] = t_{se}[t]$. Finally, by Lemma 7.8, (12) and (15) imply $t_{se}[t] = t$.

(17) $s\ F_{s/i}\ r\ R_{si}\ w$, by (12) and (16).

(18) $t_{se}[r]\ R'_{si}\ v$. By Lemma 7.7 and (17), $t_{se}[r]\ R_{si}\ t_{ie}[w]$. Since $t_{se}[r] \in D'_{s/i}$ and $t_{ie}[w] \in D'_{i/s}$, then $t_{se}[r]\ R'_{si}\ t_{ie}[w]$. From (2), $t_{ie}[w] = t_{ie}[v]$, and from (1), $t_{ie}[v] = v$.

(19) $s\ F'_{s/i}\ t_{se}[r]$. From (15), (16), and (17) by Theorem 7.10.

(20) $s\ F'_{s/i} R'_{si}\ v$, by (18) and (19). **QED.**

PROOF OF THEOREM 7.13: To prove Eq. (7.26), consider an arbitrary state pair $\langle s,v \rangle$ such that $s\ R'_{si} F'_{i/s}\ v$. It will suffice to show that if $s \notin D'^x_{s/i}$, then $s\ F'_{s/i} R'_{si}\ v$. Since $s\ R'_{si} F'_{i/s}\ v$, there exists a state u such that $s\ R'_{si}\ u\ F'_{i/s}\ v$. There are two cases; either u and v are in the same equivalence class or they are not. Lemmas 7.16 and 7.17 show that $s\ F'_{s/i} R'_{si}\ v$ for each of these cases.

Eq. (7.27) follows directly from Eq. (7.15) and Lemma 7.15. **QED.**

7.11. Hierarchical Verification

A desirable characteristic of a criterion for consistency is the ability to apply it hierarchically. That is, if program P_i is consistent with P_s, and a third program P_l is consistent with P_i, we would like to conclude that P_l is therefore consistent with P_s. The following theorem allows such a conclusion, using the implied specification Eqs. (7.14) and (7.15).

Theorem 7.18: Given the following:

- Programs $P_s = \langle D_s, F_s \rangle$, $P_i = \langle D_i, F_i \rangle$, and $P_l = \langle D_l, F_l \rangle$, and relations $R_{si} \subseteq D_s \times D_i$ and $R_{il} \subseteq D_i \times D_l$.
- Relation R_{sl} defined by

$R_{sI} = R_{st}R_{iI}$. (7.28)

- Programs $P_{s/t}$, $P_{i/s}$, $P_{i/t}$, $P_{I/t}$, $P_{s/I}$, and $P_{I/s}$ defined by Eq. (7.6).
- Corresponding don't-care vertex sets defined by Eq. (7.5).
- Programs satisfy the following conditions:

$R_{iI}F_{I/t} \subseteq (F_{i/t}R_{iI}) \cup (D^x_{i/I} \times D_{I/t})$. (7.29)

$R_{It}(D^x_{I/t}) \subseteq D^x_{i/I}$. (7.30)

$R_{sI}F_{I/s} \subseteq (F_{s/t}R_{sI}) \cup (D^x_{s/I} \times D_{i/s})$. (7.31)

$R_{is}(D^x_{i/s}) \subseteq D^x_{s/t}$. (7.32)

$D_{i/s} \subseteq D_{i/I}$. (7.33)

$\forall u \in D_i, t \in D_s:\ u\ R_{iI}R_{Is}\ t \supset (\exists s \in D_s \mid u\ R_{is}\ s)$. (7.34)

Then the following result holds:

$R_{sI}F_{I/s} \subseteq (F_{s/I}R_{sI}) \cup (D^x_{s/I} \times D_{I/s})$. (7.35)

$R_{Is}(D^x_{I/s}) \subseteq D^x_{s/I}$. □ (7.36)

Eq. (7.33) intuitively requires that no detail in $F_{s/i}$ be lost by R_{iI}. It is also required to establish the result. (A weaker form similar to Eq. (7.34) might suffice.) Figure 7-19 shows a simple example that satisfies all the conditions except Eq. (7.33), but that fails to satisfy Eq. (7.35). In particular, vertex v does not satisfy Eq. (7.33), and $\langle s,x \rangle$ is in the LHS but not the RHS of Eq. (7.35).

Figure 7-20a illustrates Eq. (7.34). The existence of the solid edges implies the existence of the dashed edge for some vertex s. This condition is also required to establish the result. Figure 7-20b shows an example that satisfies all the conditions except Eq. (7.34), but that fails to satisfy Eq. (7.35). In particular, vertex u does not satisfy Eq. (7.34), and $\langle s,x \rangle$ is in the LHS but not the RHS of Eq. (7.35).

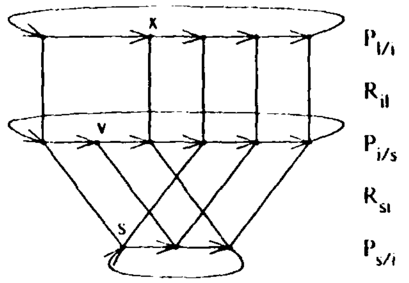


Figure 7-19: Need for Eq. (7.33)

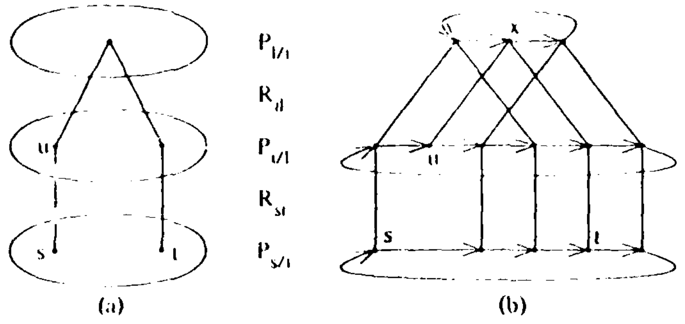


Figure 7-20: Need for Eq. (7.34)

We now present some lemmas. The lemmas assume the conditions given in Theorem 7.18. The proof of the theorem follo he lemmas.

Lemma 7.19: $D_{s/t} = D_{s/I}$. □

PROOF: $D_{s/l} \subseteq D_{s/i}$: Given a vertex s in D_s, if $s\ R_{sl}\ x$ for any vertex x, then by Eq. (7.28), $s\ R_{si}\ u$ for some vertex u. (A similar argument also shows $D_{l/s} \subseteq D_{l/i}$.)

$D_{s/i} \subseteq D_{s/l}$: Prove by contradiction. Assume there is a vertex s in D_s such that s is in $D_{s/i}$ but not $D_{s/l}$. Then $s\ R_{si}\ u$ for some vertex u in D_i. This implies that u is in $D_{i/s}$ but not in $D_{i/l}$; if u were in $D_{i/l}$ then s would be in $D_{s/l}$. This contradicts Eq. (7.33). **QED.**

Lemma 7.20: $(D^x_{i/l} \cap D_{i/s}) \subseteq D^x_{i/s}$. □

PROOF: Show by contradiction. Suppose $u \in (D^x_{i/l} \cap D_{i/s})$, but $u \notin D^x_{i/s}$. Since u is in $D_{i/s}$ but not $D^x_{i/s}$, then $u\ F_i+\ v \in D_{i/s}$ for some vertex v. Now $D_{i/s} \subseteq D_{i/l}$ by Eq (7.33), so $\{u,v\} \subseteq D_{i/l}$. Therefore $u\ F_{i/l}+\ v$. This contradicts $u \in D^x_{i/l}$. **QED.**

Figure 7-21 illustrates the following lemma. This lemma states conditions under which the existence of a path in P_l implies the existence of a related path in P_i.

Lemma 7.21: Given vertices $u \in D_{i/s}$, $w \in D_{l/s}$, and $x \in D_{l/i}$ such that $u \notin D^x_{i/s}$ and $u\ R_{il}\ w\ F_l^*\ x$ along a path containing no vertices in $D_{l/s}$ except w and possibly x. Then there exists a vertex v in $D_{i/l}$ such that $u\ F_i^*\ v\ R_{il}\ x$ along a path containing no vertices in $D_{i/s}$ except u and possibly v. We have $v \in D_{i/s}$ if and only if $x \in D_{l/s}$. □

PROOF: Assume vertices u, w, and x as given. If $x = w$, then the result is immediate with $v = u$. Assume $x \neq w$. From Eq. (7.28) we know $D_{l/s} \subseteq D_{l/i}$, so

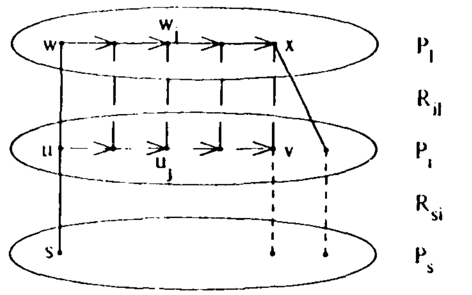


Figure 7-21: Lemma asserting existence of path in P_l

$w \in D_{l/i}$. Then $w\ F_{l/i}+\ x$ along a path containing no vertices in $D_{l/s}$ except w and possibly x. Let $w_0, w_1, \ldots, w_n$ be the sequence of vertices w_j in such a path, with $n \geq 1$, $w_0 = w$, $w_n = x$. We show by induction that we can find vertices $u_1, \ldots, u_n$ such that for $0 < j \leq n$, $u\ F_i+\ u_j\ R_{il}\ w_j$ along a path containing no vertices in $D_{i/s}$ except u (and u_n *iff* $j = n$ and $w_n \in D_{l/s}$).

First consider $j = 1$. Lemma 7.20 implies $u \notin D^x_{i/l}$, since $u \in D_{i/s}$ but $u \notin D^x_{i/s}$. Then Eq. (7.29) gives $u\ F_i+\ u_1\ R_{il}\ w_1$ for some vertex u_1, along a path containing no vertices in $D_{i/l}$ except u and u_1. Since $D_{i/s} \subseteq D_{i/l}$ (Eq. (7.33)), only u and u_1 in this path could be in $D_{i/s}$. If $n = 1$ and $w_n \in D_{l/s}$, then Eq. (7.34) implies that $u_1 \in D_{i/s}$. If $n > 1$ or $w_n \notin D_{l/s}$, then $u_1 \notin D_{i/s}$; otherwise Eq. (7.28) would give $w_1 \in D_{l/s}$, a contradiction.

Now assume the inductive assertion is true for $j = j0$ and consider $j = j0 + 1 \leq n$. Suppose $u_{j0} \in D^x_{i/l}$. Then there is a don't-care vertex z in D^x_i such that $u_{j0}\ F_i^*\ z$ along a path not including any vertices in $D_{i/l}$ except u_{j0}. Since

$D_{l/s} \subseteq D_{l/l}$, the path also includes no vertices in $D_{l/s}$ except possibly u_{j0}, but $u_{j0} \notin D_{l/s}$ since $j0 < n$. Then from the inductive assertion, $u\ F_l + z$ along a path including no vertices in $D_{l/s}$ except u. This implies $u \in D^x_{l/s}$, a contradiction. Therefore $u_{j0} \notin D^x_{l/l}$.

Now Eq. (7.29) implies $u_{j0}\ F_l + u_{j0+1}\ R_{ll}\ w_{j0+1}$, for some vertex u_{j0+1}, along a path containing no vertices $D_{l/l}$ except u_{j0} and u_{j0+1}. From the argument at the end of the preceding paragraph, $u_{j0+1} \in D_{l/s}$ if and only if $n = j0 + 1$ with $w_n \in D_{l/s}$.

This completes the induction. The inductive premise satisfies the lemma for $j = n$ with $v = u_n$. **QED.**

Lemma 7.22: $R_{ll}(D^x_{l/s}) \subseteq D^x_{l/s}$. □

PROOF: Consider arbitrary vertices w and u such that $w \in D^x_{l/s}$ and $w\ R_{ll}\ u$. It will suffice to show that $u \in D^x_{l/s}$. Since $w \in D^x_{l/s}$, there exists vertex $y \in D^x_l$ such that $w\ F_l^*\ y$ along a path containing no vertices in $D_{l/s}$ except w. Let x be the last vertex of this path that is in $D_{l/l}$. Now $x \in D^x_{l/l}$.

The proof will be by contradiction. Eq. (7.34) implies $u \in D_{l/s}$. Suppose $u \notin D^x_{l/s}$. Then by Lemma 7.21, there exists v such that $u\ F_l^*\ v\ R_{ll}\ x$ along a path containing no vertices in $D_{l/s}$ except u. By Eq. (7.30), $v \in D^x_{l/l}$, so there exists $z \in D^x_l$ such that $v\ F_l^*\ z$ along a path containing no vertices in $D_{l/l}$ except v. Since $D_{l/s} \subseteq D_{l/l}$ (Eq. (7.33)), we now have $u\ F_l^*\ z$ along a path containing no vertices in $D_{l/s}$ except u. Therefore $u \in D^x_{l/s}$. This contradicts the supposition and proves the desired result. **QED.**

PROOF OF THEOREM 7.18: To prove Eq. (7.35),

$$R_{sl}F_{l/s} \subseteq (F_{s/l}R_{sl}) \cup (D^x_{s/l} \times D_{l/s}) \quad , \tag{7.35}$$

make the following observations.

(1) $s\ R_{sl}F_{l/s}\ x$, where $s \notin D^x_{s/l}$, by assumption. It will suffice to show that $s\ F_{s/l}R_{sl}\ x$.

(2) $s\ R_{sl}\ u\ R_{ll}\ w\ F_{l/s}\ x$ for some vertices u and w, from (1) by Eq. (7.28).

(3) $u \notin D^x_{l/s}$. Otherwise, Eq. (7.32) would imply $s \in D^x_{s/l}$. This would contradict (1), since Lemma 7.19 and Eqs. (7.6) and (7.5) imply $D^x_{s/l} = D^x_{s/l}$.

(4) $u\ F_{l/s}\ v\ R_{ll}\ x$ for some vertex v, from (2) and (3) by Lemma 7.21, with $v \in D_{l/s}$ since $x \in D_{l/s}$.

(5) $s\ F_{s/l}\ t\ R_{sl}\ v$ for some vertex t, from (2) and (4) by Eq. (7.31), since $s \notin D^x_{s/l}$ (see (3)).

(6) $s\ F_{s/l}R_{sl}\ x$, from (4), (5), and Eq. (7.28). Lemma 7.19 and Eq. (7.6) imply $F_{s/l} = F_{s/l}$.

This proves Eq. (7.35). Eq. (7.36) follows from Lemmas 7.19 and 7.22 and Eqs. (7.28) and (7.32). **QED.**

Theorem 7.18 says that if program P_l is consistent with [illegible], which in turn is consistent with P_s according to Eqs. (7.14) and (7.15), then we may conclude that P_l is consistent with P_s without comparing P_l against P_s. This result holds provided that relations R_{sl} and R_{ll} satisfy the conditions in Eqs. (7.33) and (7.34).

7.12. Summary

Milner and Brand proposed definitions of program consistency for two particular types of program models, deterministic and nondeterministic programs without don't-cares. This chapter considered these model types plus the corresponding models with don't-cares. Section 7.8 gave a *criterion* for consistency suitable for all four models and compatible with the definitions proposed by Milner and Brand.

Brand argued for the inclusion of closure in the criterion for practical reasons. We saw *that this leads to the verification of clearly incorrect* designs, but the criterion without closure is difficult to test. Section 7.9 added closure to the criterion in a limited way. Section 7.10 showed the correspondence between this new criterion and the simple criterion for deterministic models.

Finally, Section 7.11 described hierarchical verification. Milner defined a "strong simulation" consistency criterion that he described as a transitive reflexive relation leading to a "quasi-ordering" of programs. This notion is similar to that of hierarchical verification, but applies only to deterministic models without don't-cares, with all inputs presented before execution, and execution terminating with a [illegible] output value. That model is not suitable for hardware descriptions. [illegible] 7.11 showed that hierarchical verification is possible for more general [illegible] models.

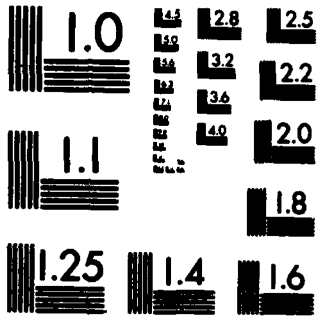
[illegible] of problems remain to be solved to achieve a unified, useful theory [illegible]. These problems are outlined in Chapter 9.

AD-A136 466 VERIFICATION OF HARDWARE DESIGN CORRECTNESS: SYMBOLIC EXECUTION TECHNIQUE..(U) STANFORD UNIV CA COMPUTER SYSTEMS LAB W E CORY JUN 83 TR-83-241 DAAG29-80-K-0046

2/2

UNCLASSIFIED F/G 9/2 NI

END
DATE FILMED
2-84
DTIC


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

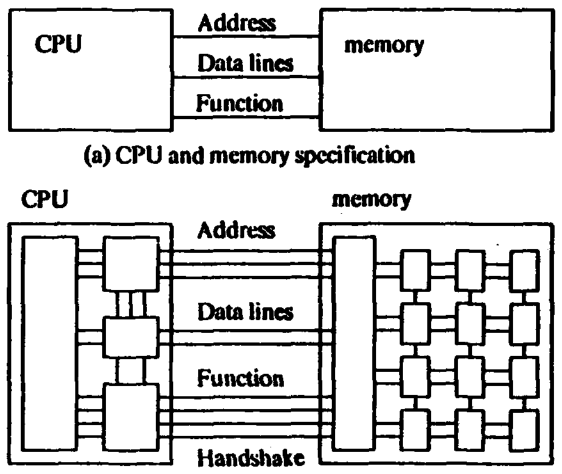
Chapter 8
Partitioned Verification

A hardware specification to be compared against a proposed implementation may consist of several interconnected components. In that case, it is desirable to verify the design for each component individually.

For example, consider Figure 8-1. Part (a) represents a system specification with two components, a CPU and a memory. Part (b) represents a proposed design for this system. We may verify the CPU design ignoring the memory component, and then similarly verify the memory design. The verification of each component individually is a partitioned verification. This is especially desirable if one component type is used many times, or if the design is specified hierarchically.

The implicit assumption in partitioned verification is that if each individual component design is correct, then the combination of these designs is also correct. Brand demonstrated this property under limited conditions [Br78]. This thesis will not expand on this theoretical base, although a number of improvements are needed as mentioned in Chapter 9. However, this chapter will discuss the impact of the requirements for Brand's theorem.

Section 8.1 reviews Brand's result. Section 8.2 presents an example that



(a) CPU and memory specification

(b) CPU and memory design

Figure 8-1: Changes in data type

demonstrates the characteristic of a design that may prevent the designer from satisfying the requirements of this theorem. Finally, Section 8.3 describes a modification of the design process that allows the requirements to be satisfied. This modification uses translators, a concept proposed by Hill [Hi80].

8.1. Review of Partitioned Verification

This section repeats Brand's theorem for partitioned verification. No new proofs are presented; the purpose of this section is to explain in more detail what partitioned verification is and to highlight the restrictions under which the theorem holds.

This theorem allows partitioned verification for a design with two components. This result easily extends to an indefinite number of components, and we state the result below for the general case. For brevity, we omit the elegant notion of extending relations in the development of the program model.

In the following discussion, the definition of "referring to Δ" (Section 8.1.2) is stronger than Brand's. We also assume a restriction from Section 7.5 not required by Brand. These changes ar · required to repair a minor error in Brand's proof. We do not repeat or correct that proof here. The interested reader may consult Appendix B, which discusses the correction of the proof. The appendix assumes familiarity with Brand's technical report [Br78].

8.1.1. Brand's Program Model

The program (hardware description) consists of the combination of n components. Intuitively, a component is an autonomous block specified separately from the other blocks. We use graphs to represent programs, as in Chapter 7. Assume nondeterministic models with no don't-cares.

In the graph model, component j is a program $P_a^j = \langle D_a, F_a^j \rangle$. (Recall that subscript "a" denotes program "s" or "i".) By convention, a superscript will identify a component. Then P_a is the union of the components; $P_a = \langle D_a, \cup_{j=1}^{n} F_a^j \rangle$. The state space D_a for program P_a is split among the n components:

$$D_a = \Delta_a \times D_a^1 \times D_a^2 \times \dots \times D_a^n \quad .$$

Then D_a^i is the set of local variables for component i, while Δ_a is the set of variables shared by the components. If $s \in D_a$, then s^i is the element of s in D_a^i; s^0 is the element of s in Δ_a. Also, $s(v \backslash i)$ will denote state s with element s^i set to v.

F_a^i describes the actions of component i, and so can modify only its own local variables in D_a^i and the shared variables in Δ_a. These actions are independent of the values of the local variables for any other component. The following assertion expresses these properties:

$\forall$ states s, t and integers $i, j \mid s\, F_a^i\, t$:

$$((s^j \neq t^j) \supset (j = i \text{ or } j = 0)) \text{ and}$$
$$((j \neq i \text{ and } j \neq 0) \supset (\forall v \in D_a^j\colon s(v \backslash j)\, F_a^i\, t(v \backslash j))) \quad .$$

Finally, relation R_{ab} is defined in a piecewise fashion. We have $R_{ab}^i \subseteq D_a^i \times D_b^i$; also $R_{ab}^0 \subseteq \Delta_a \times \Delta_b$. Then $s\, R_{ab}\, t$ if and only if, for $0 \leq i \leq n$, $s^i\, R_{ab}^i\, t^i$. A notation to express this is

$$R_{ab} = R_{ab}^0 \,\&\, R_{ab}^1 \,\&\, \dots \,\&\, R_{ab}^n = \&_{i=0}^{n} R_{ab}^i \quad .$$

Now $D_{a/b}$, $D_{a/b}^i$, $F_{a/b}$, etc. are defined by Eq. (7.6). We assume the requirement from Section 7.5 for all the edge sets F_a^i; namely, every path in $P_a^i = \langle D_a, F_a^i \rangle$ leading from any vertex in $D_{a/b}$ must eventually reach a vertex in $D_{a/b}$.

8.1.2. Definitions

A "stopping point of $D_{a/b}$ for component i" is a state s with $s^i \in D_{a/b}^i$ and $s^0 \in \Delta_{a/b}$. Note that a state in $D_{a/b}$ is a stopping point of $D_{a/b}$ for every component.

A transition $s\, F_a^i\, t$ "refers to Δ_a" if there exist $r \in D_a$ and $d_1, d_2 \in \Delta_a$ such that $s(d_1 \backslash 0)\, F_a^i\, r$ and either (i) $d_1 \neq r^0$ or (ii) $\langle s(d_2 \backslash 0), r(d_2 \backslash 0) \rangle \notin F_a^i$.

We say "there is a stopping point of $D_{a/b}$ for component i between any two references to Δ_a by F_a^i" if whenever

$s_0 F^i_a s_1 F^i_a \ldots F^i_a s_m F^i_a s_{m+1}$

($m \geq 1$), where the first and last transitions refer to Δ_a, then some s_k in the path, $1 \leq k \leq m$, is a stopping point of $D_{a/b}$ for component i.

8.1.3. Theorem for Partitioned Verification

Brand's theorem, with the minor changes noted above, is as follows.

Theorem 8.1: Assume partitioned, nondeterministic programs P_s and P_i without don't-cares, each with n components, and relation $R_{si} \subseteq D_s \times D_i$, all as described in Section 8.1.1. Suppose the following conditions hold for j, $1 \leq j \leq n$:

- $R_{si} F^j_{i/s}{}^* \subseteq F^j_{s/i}{}^* R_{si}$.
- there is a stopping point of $D_{a/b}$ for component j between any two references to Δ_a by F^j_a.

Then the following result holds:

$R_{si} F_{i/s}{}^* \subseteq F_{s/i}{}^* R_{si}$. □

One restriction above is that any two successive references to Δ_a must be separated by a stopping point. The other restriction is that every path F^j_a+ starting from a stopping point of $D_{a/b}$ for component j must eventually reach another such (or the same) stopping point.

It is likely that any theory allowing partitioned verification will include these restrictions in some form. The requirement that every path from a stopping point reaches a stopping point ensures that $P_{a/b}$ adequately represents the behavior of P_a.

The requirement that a stopping point separates two references to Δ_a ensures that protocols for communication between components are strongly enough defined to permit the independent verification of components. Each of these properties is important for partitioned verification.

The example in the next section illustrates the problem these restrictions cause in a hierarchical design.

8.2. Partitioned Verification Problem

In a hierarchical design, different descriptions may use different levels of abstraction. This means that two descriptions to be compared may use different representations of the same data. If the change in the level of abstraction affects only the static encoding of data, then a simulation relation may readily satisfy the restrictions for Brand's theorem. However, the introduction of new timing details may hinder the fulfillment of these requirements.

Consider the example in Figure 8-1. Part (a) represents a specification of a CPU, memory, and bus at a high level of abstraction. Details of the bus protocol may be omitted or simplified. *In contrast, the refined description in* Figure 8-1b may specify the bus structure and protocol in greater detail.

Figure 8-2 shows the timing of a write to memory for these two descriptions. The *CPU Ctrl* is an internal control variable in the CPU. *Mem Data* is the contents of the memory being written. The *Address* and *Data* lines are controlled by the CPU. *Write Ctrl*, *Strobe*, and *Ack* are parts of the *Function* lines in the two descriptions. Note the labeling of states by numbers in part (a) and letters in part (b).

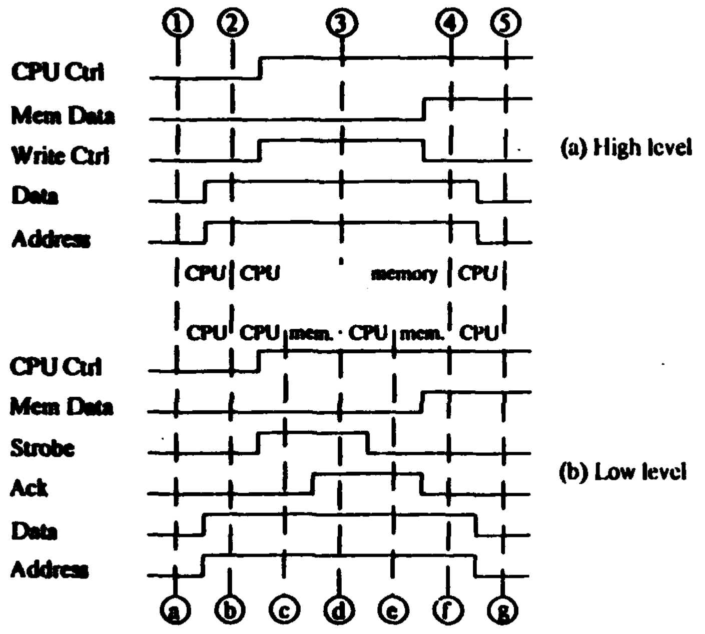


Figure 8-2: Timing diagrams for write cycle

The specification (part (a)) uses simple timing. The *Write Ctrl* does not behave like a bus line in the usual sense; the CPU asserts it and the memory negates it, so it behaves more like a latch. This does not necessarily dictate how the write cycle will be implemented. In the design (part (b)), the CPU controls *Strobe* and the memory controls *Ack*.

Now Figures 8-3 and 8-4 show the graph models for the write cycles in

Figure 8-2. The state labels are taken from Figure 8-2. It is desirable to verify Figure 8-4a against Figure 8-3a and Figure 8-4b against Figure 8-3b, and so verify the design. Of course, we need a simulation relation.

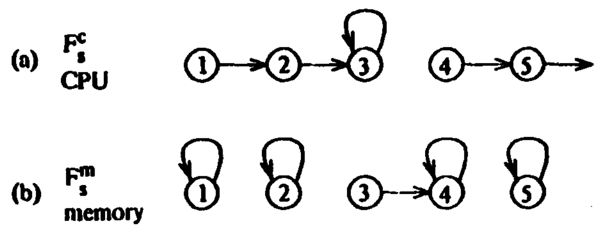


Figure 8-3: Graph models for high-level write cycle

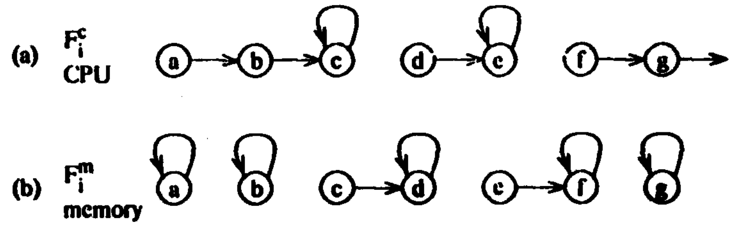


Figure 8-4: Graph models for low-level write cycle

States (1), (2), (4), and (5) naturally correspond to states (a), (b), (f), and (g), respectively. This leaves state (3) to correspond to (c), (d), or (e). But each of these choices is unsatisfactory, resulting in incomplete tests for correctness. For example, if we let (3) R_{si} (d), then the verification process does not check the behavior of F_i^c at (b), (c), (d), or (e), and it does not test F_i^m at (c) or (e).

Letting (3) correspond to combinations of states (c), (d), and (e) does not improve matters. For example, assume $R_{si}((3)) = \{(c),(d),(e)\}$. Now suppose that through a design error, we have (d) F_i^c (d) instead of (d) F_i^c (e). The resulting design would not operate correctly, but the application of either Eq. (7.4) or (7.7) to $F^c_{s/i}$, $F^c_{i/s}$, and R_{si} would flag no error.

The problem here is that program P_i uses finer timing detail than P_s does. For such cases, a simulation relation is likely either *to not satisfy the restrictions* for Theorem 8.1 or to not distinguish between correct and incorrect actions in the implementation. In the above example, the former occurred when state (3) corresponded to just one of (c), (d), and (e); the latter occurred when (3) corresponded to all three states.

The use of the same timing detail in two descriptions being compared facilitates the fulfillment of the restrictions for Theorem 8.1 by an appropriate simulation relation. A modification to the top-down design methodology can ensure this correspondence. In this new method, the designer will still have the freedom to use varying levels of abstraction. However, the design will be refined in such a way that any two descriptions to be compared will always use the same timing. Partitioned verification may then be possible.

8.3. Modification of the Design Methodology

In a top-down design methodology, each refinement may introduce finer structural and behavioral detail. As noted above, the addition of new timing detail at some stage of the design may hinder the fulfillment of the restrictions for

partitioned verification. The design methodology may be expanded to handle this problem.

The expanded methodology is presented below. Block diagrams are used for illustration. Blocks represent components. Lines interconnecting the blocks represent variables shared by the components.

The first step in the expanded methodology is to write a behavioral specification for a system to be designed. This is represented in Figure 8-5. This system consists of several components, and it is desirable to verify the design for this system in a partitioned fashion. Due to the difficulties discussed in the previous section, this may not be possible if the next refinement introduces new timing details.

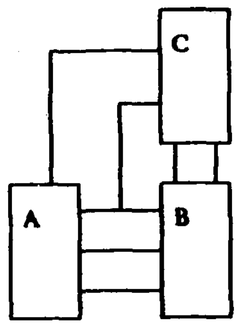


Figure 8-5: Initial system specification

To overcome this problem, if the designer wishes to use more detailed timing in the next refinement, then he must first alter the specification by the addition of translators. A translator is similar to a component description, but its significance differs. While a component description represents the behavior of some hardware, a translator relates timing behavior at one level of abstraction to that at another level. Figure 8-6 shows the original specification with translators added. The translators are represented by triangles.

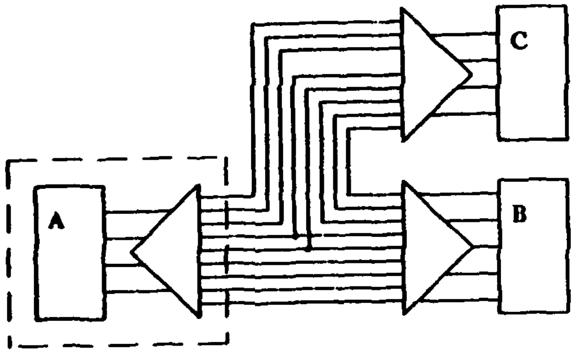


Figure 8-6: Specification with translators

The next step is to verify the correctness of the translators. The criterion for correctness is that the presence or absence of translators should be transparent in the operation of the system. The verification step consists of showing that the description in Figure 8-6 (P_i for this step) is consistent with the specification ("P_s") in Figure 8-5.

The components in Figure 8-5 use high-level timing. After defining and verifying the translators, new component specifications using new low-level timing

may be taken from Figure 8-6. For example, the circuit inside the dotted box is a new specification for component A. It uses low-level timing for its accesses to variables shared with other components.

The designer now makes the next refinement in the design. Figure 8-7 represents the design for component A. Since this design and the specification from Figure 8-6 use the same timing for accesses to variables shared with other components, the designer should now be able to define a simulation relation satisfying the restrictions for partitioned verification. The design in Figure 8-7 may be verified separately from the designs for the other components.

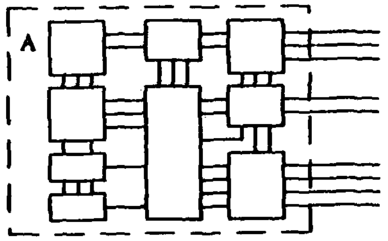


Figure 8-7: Interface design refinement

8.4. Summary

Given a behavioral specification *P* for a system to be designed, the expanded top-down design methodology may be summarized as follows.

If the next refinement in the design will use more detailed timing, then define translators that convert the old high-level timing into the new timing. Insert these translators into P, yielding description Q in which components communicate with each other using the new timing. Verify Q against P.

Now generate design R_q for each component q in Q. The combination of designs R_q yields design R. Define a simulation relation between descriptions R and Q that satisfies the requirements for partitioned verification. Since the descriptions R and Q use the same timing, this should be possible. Verify R against Q. This can be done in a piecewise fashion; verify R_q against q for each component q.

Repeat the above cycle for each design R_q until the lowest-level design uses available components or components to be designed using other design automation tools.

This is quite similar to the original top-down design procedure. The only change is the definition and insertion of translators when needed. If the designer does not wish to introduce new timing details in some refinement of the design, then descriptions P and Q are identical. In that case, the new methodology is identical with the old. Hence, the new design style preserves the flexibility of the old style.

The new methodology is similar to a common approach in the design of systems with components that communicate over a bus. The definition of translators corresponds to the definition of a bus protocol. The design for each component is then validated against its specification and the bus protocol. The designer assumes that components which observe the bus protocol will then communicate with each other properly.

The new design methodology does require some extra work on the part of the designer; namely, the specification of translators which define the refinements in timing. However, the new methodology does not make new work where no work existed before. At some point in the old methodology, the designer had to define the refinements in timing before using them. The new methodology simply requires that these refinements be expressed formally in translator definitions.

Hill defined translators in ADLIB to facilitate multi-level simulation [Hi80]. The use of translators to facilitate partitioned verification represents a new application of this concept.

Chapter 9
Summary

9.1. Contributions

This thesis investigates two problems: reduction of case splitting in the symbolic execution of event-driven simulations, and the criterion for the consistency of two hardware descriptions in functional verification.

9.1.1. Non-selective Trace

An event-driven simulation consists of a collection of independent components with some means of communication between them. The simulator commonly will not simulate a component unless the value of a particular variable(s) changes. This is called "selective trace" [Ul65]. When the variable in question receives a new value, the simulator must test whether the new value is different from the old to determine whether to simulate the component. If a symbolic execution cannot uniquely resolve this test to *TRUE* or *FALSE*, then the single execution will split into several executions (cases), one for each branch followed [HK76, Ki76a].

The presence of distinct, independent components aggravates this problem. If each of I components always splits the simulation into N cases in every time step, then after J time steps, the total number of cases will be N^{IJ}.

Chapter 4 suggests the substitution of non-selective trace for selective trace to eliminate these case splits. The non-selective trace simulates a component whenever the proper variable is written, whether or not the operation changes the value. Of course, this substitution is correct only if the non-selective version is functionally equivalent to the selective version.

In the model for event-driven simulation given in Chapter 3, the substitution of non-selective trace for selective trace is correct for the construction typically used to describe combinational elements. More generally, consider a component described by an infinite loop that waits at one point for an event. Under certain conditions, the use of non-selective trace is appropriate for such loops. The major condition is the following: If the main body of the loop were executed even though no event occurred, then it would not modify any variables.

The annotation of a loop with assertions can facilitate a proof that the loop satisfies the conditions for use of non-selective trace. An inductive assertion analysis in turn proves the assertions. However, the assertions are often cumbersome and *redundant*. Section 4.2.6 gives a *sufficient* test that checks the loop without the formulation of the redundant assertions. This test symbolically executes the body of the loop twice in succession under certain assumptions, then checks that the second execution did not alter any variables.

Previous work did not apply symbolic execution to event-driven simulation. In particular, it did not investigate the substitution of non-selective trace for selective trace to reduce case splitting.

9.1.2. Case Merging

Conditional expressions can symbolically express the computational results of several parallel paths [CH79]. For example, the expression

$C1$ and $C2$ or not $C1$ and $C3$

gives the value of the Boolean variable X after

```
if C1 then X := C2
     else X := C3;
```

regardless of the path selected in the if statement. A flow graph similar to that used by Aho and Ullman for dataflow analysis [AU77] directs the generation of the conditional expressions.

In the simulation model of Chapter 3, each component is represented by a coroutine. The following steps facilitate the use of conditional expressions to combine the results of multiple paths in coroutines.

1. Allow the initial and final values of a coroutine's program counter to be symbolic (not constant).
2. Do a topological sort of a coroutine's flow graph to determine the order in which the coroutine's basic blocks should be processed.

Using these extensions, Section 5.2.2 describes an analysis of a hardware description yielding a new equivalent description. The new description contains no conditional branches that would introduce case splits in a symbolic execution. This allows a symbolic execution that combines all parallel paths into a single execution; hence the term "case merging."

This construction works for components in which every loop is broken by a

statement that waits for an event to occur. A modification extends the construction to apply to components with unbroken loops if the loops unfold.

The major problem with case merging is the potentially excessive complexity of the symbolic expressions generated in a symbolic execution. Section 5.4 identifies a number of optimizations that will simplify the symbolic expressions. Chapter 6 considers the tradeoff between the reduction in the number of cases and the increase in symbolic expression complexity. Case merging is most likely to be profitable for functional verification when the simulation relation is simple and the expression complexity grows less than linearly as the number of cases in a case splitting execution.

Cheatham et al. first proposed the use of conditional expressions to symbolically express the results of parallel paths. They applied this technique in a static analysis of programs including procedures but not coroutines. The contributions of Chapters 5 and 6 are (i) enhancements of case merging for coroutines, (ii) alternative treatment of unfolding loops, (iii) optimizations, and (iv) evaluation of usefulness of case merging for functional verification.

9.1.3. Consistency of Hardware Descriptions

The functional verification problem is that of comparing two hardware descriptions to determine their consistency. One description is a specification while the other is a proposed design. Milner and Brand considered similar questions, representing hardware descriptions (or programs in Milner's case) by graph models [Mi71, Br78]. This thesis extends their work.

Milner assumed a deterministic model with no don't-cares. Brand assumed a nondeterministic model, also without don't-cares. Section 7.8 presents a simple criterion for consistency that applies to both deterministic and nondeterministic models, with or without don't-cares. ("Simple" distinguishes this criterion from the extended criterion below.)

In some cases, [illegible] description may specify an action by a single operation while the other d[illegible]ption uses a loop to specify the same action. Brand's criterion included an exten[illegible] to facilitate the verification of such cases by symbolic execution. However, [illegible]is extension allows the verification of incorrect designs. Section 7.9 presents an alternate extension that also facilitates verification of descriptions with loops.

For deterministic models, Section 7.10 demonstrates the compatibility of this new extension with the new simple criterion from Section 7.8. This supports a claim that the new extension retains intuitive notions of correctness.

Milner demonstrated that under certain conditions, his criterion of consistency is transitive, a desirable property for hierarchical hardware verification. However, his model was tailored for software, not hardware. He partitioned the vertex (data) space and specified restrictions on the simulation relation relating the two graph models. He also required all inputs to be presented before execution started. These restrictions are not appropriate for hardware descriptions. Section 7.11 proves the transitivity of the new simple criterion under less restrictive conditions.

Finally, Brand proved a theorem allowing verification of a design in a

piecewise fashion. One of the conditions for this theorem prevents piecewise verification of a design if it contains timing details not included in the specification. Chapter 8 modifies the hierarchical design methodology to permit piecewise verification of a refinement that introduces new timing details. The new methodology specifies that before making such a refinement, we modify the specification so that it uses the same timing as the proposed refinement.

9.2. Suggestions for Further Research

9.2.1. Case Merging

It is desirable to obtain more definitive results on the merits of case merging vs. case splitting. The effects of the choice of canonical form, simplification, or efficient expression comparison heuristics make such an analysis difficult.

9.2.2. Criteria for Consistency

Ideally, we would have one criterion for consistency which achieves all of the following:

1. applies to deterministic and nondeterministic models with and without don't-cares,
2. facilitates verification of descriptions with loops,
3. supports hierarchical and partitioned verification.

However, we now have three criteria: the new simple test, new extended test, and Brand's test. These tests have the following properties:

- The new extended test satisfies (1) and (2) above, but (3) has not been demonstrated.

- The new simple test supports hierarchical verification.
- Brand's test supports partitioned verification, but only for nondeterministic models.
- For deterministic models, the new simple and extended tests are in some sense compatible.

It is desirable to close the gap between the achieved theory and the ideal, unified theory. Some possible approaches follow:

1. Prove partitioned verification for the new simple test with deterministic models. Use compatibility of simple and extended tests to prove hierarchical and partitioned verification for the extended test. This unifies the consistency theory for deterministic models.

2. Prove partitioned verification for the new simple test with nondeterministic models. Prove compatibility of simple and extended tests for nondeterministic models. Use this compatibility to prove hierarchical and partitioned verification for the extended test. This unifies the theory for nondeterministic models.

3. Prove hierarchical and partitioned verification for the new extended test with a particular model. This unifies the theory for the selected model.

4. Find conditions under which Brand's test is equivalent to the extended test. Prove hierarchical verification for Brand's test with nondeterministic models. This unifies the theory for nondeterministic models, with conditions that determine whether Brand's test is appropriate.

Brand's result for partitioned verification allows only one resource to be shared between the several components, with only one reference to the resource between stopping points in the simulation relation. This is likely to be impractical in hierarchical hardware designs. Relaxation of this restriction is desirable.

9.2.3. Symbolic Execution for Functional Verification

This thesis considered symbolic execution separately from the definition of consistency for functional verification. Development is required on the interpretation of symbolic execution results to verify consistency. One problem is to guarantee that the symbolic executions adequately cover the possible input sequences.

9.2.4. Incorporation of Other Techniques for Verification

In some cases, symbolic execution may not be a suitable technique for comparing two hardware descriptions. The correctness of a particular component in a design may rely on significant mathematical theory not easily incorporated into symbolic execution theorem provers. For example, the comparison by symbolic execution of an arithmetic description of a 32-bit adder with a Boolean description of the same adder is difficult, due to the complex Boolean expressions generated. This problem is compounded for floating point hardware. Nonetheless, designs known to be correct are available, and the designer should be free to use them. In such cases, it is desirable to be able to establish the correctness of these components by independent means, and then to incorporate these results for the remainder of the proof.

9.2.5. Strengthening the Weak Links

'n a top-down design, the designer generates a chain of hardware descriptions, starting with the first formal specification of the system behavior, and ending with a formal description of the physical implementation. Each verification of two successive descriptions forms a link in the verification chain ensuring the functional correctness of the design. However, two special cases exist in which one of the two descriptions being compared is not a formal description. These special cases are the two ends of the chain.

To have confidence in the proof of correctness, the designer must be able to show that (1) the first formal specification accurately captures the intent of the informal English specification, and that (2) the last formal specification accurately models the behavior of the physical hardware. The addition of assertions to hardware descriptions, perhaps by the application of temporal logic [Bo82, MP81], may allow the proof of desirable properties about the descriptions by performance verification. This in turn would increase the designer's confidence in the design.

Appendix A
Notation

A.1. Formulae

The expression "$a(x\backslash y)$" will denote a with the value of $a.y$ (if a is a record) or $a[y]$ (if a is an array) set to x. Vertical bar "|" in formulae may be read as "such that".

When a colon ":" appears in formulae, it will generally follow a specification of a range or set of values. The expression "a: b" then means "for each item in a, the expression or predicate b".

A.2. Relations

Given sets W and Y, a subset S of $W \times Y$ (the Cartesian product of W and Y) is called a binary relation on $W \times Y$. We write "$w\, S\, y$" as an alternate notation for "$\langle w, y \rangle \in S$".

If W, Y, and Z are sets with relations S on $W \times Y$ and T on $Y \times Z$, then ST is the "composition" of the relations:

$$ST = \{\langle w, z \rangle \in W \times Z \mid (\exists y \in Y \mid (w\, S\, y) \wedge (y\, T\, z))\} \quad .$$

The following statements are then synonymous:

$$\exists y \in Y \mid (w\, S\, y) \wedge (y\, T\, z) \quad ,$$

$$\exists y \in Y \mid w\, S\, y\, T\, z \quad ,$$

$$w\, ST\, z \quad .$$

We will also use a closure notation, given relation R on $Z \times Z$:

$$R^* = \{\langle z_s, z_e \rangle \in Z \times Z \mid$$
$$(\exists \text{ a sequence of one or more elements } z_1, z_2, \ldots, z_n \in Z \mid$$
$$z_s = z_1\, R\, z_2\, R \ldots R\, z_{n-1}\, R\, z_n = z_e)\} \quad ,$$

$$R+ = R\, R^* \quad .$$

Finally, given relation S on $W \times Y$, define the inverse relation:

$$S^{-1} = \{\langle y, w \rangle \in Y \times W \mid w\, S\, y\} \quad .$$

The image of W under S is denoted by $S(W)$:

$$S(W) = \{y \in Y \mid (\exists w \in W \mid w\, S\, y)\} \quad .$$

For an element w in W, we may write "$S(w)$" to denote $S(\{w\})$.

These conventions for relations can be used with edge sets and functions, since they may be regarded as relations.

Appendix B
Notes on Proof of Theorem 8.1

This appendix describes a correction of an error in the proof of Lemma 2 in Brand's technical report [Br78]. It is assumed the reader has access to the report. The lemma, its proof, and introductory material will not be repeated here. This discussion uses Brand's notation. The new restrictions from Section 8.1 are repeated here using Brand's notation:

Definition 1: Let $\langle D\times\Delta, C, S\times\Sigma\rangle$ be an annotated program. A transition $\langle x_1,d_1\rangle\, C\, \langle x_2,d_2\rangle$ "refers to Δ" if and only if there exist $x_3 \in D$ and $d_3,d_4,d_5 \in \Delta$ such that $\langle x_1,d_4\rangle\, C\, \langle x_3,d_3\rangle$ and either (i) $d_4 \neq d_3$ or (ii) $\langle\langle x_1,d_5\rangle, \langle x_3,d_5\rangle\rangle \notin C$. □

This definition is stronger than Brand's. It says that if some edge $\langle x_1,d_4\rangle\, C\, \langle x_3,d_3\rangle$ refers to Δ by Brand's definition, then all edges incident out of $\langle x_1,d_4\rangle$ also refer to Δ, as do all edges incident out of corresponding vertices $\langle x_1,d_1\rangle$. This makes the notion of "referring to Δ" a property more of the vertex element x_1 than of individual edges, an intuitively appealing change which facilitates the proof of Lemma 2.

Requirement 1: Consider annotated program $\langle D,C,S\rangle$. We require that every

path C^* leading from any vertex in S must eventually reach a vertex in S. This requirement applies both to C_1 and to C_2. □

B.1. Counterexamples

First we show two counterexamples that disprove Lemma 2 and show the need for the definition 1 and requirement 1 above.

Figure B-1 shows edge sets C_1 and C_2 for the first counterexample. State set D_1 is $\{0,1,2,3\}$, D_2 is $\{0\}$, and Δ is $\{0,1\}$. The stopping point sets are $S_1 = \{0,1,3\}$, $S_2 = \{0\}$, and $\Sigma = \{0,1\}$. (Brand's treatment of stopping points is different from that in Section 8.1, due to his notion of extending relations.) In the graph of C_i in Figure B-1, states with double circles are stopping points for C_i. Edges marked by a dot refer to Δ according to Brand's definition. The state name is xyd, where $x \in D_1$, $y \in D_2$, and $d \in \Delta$.

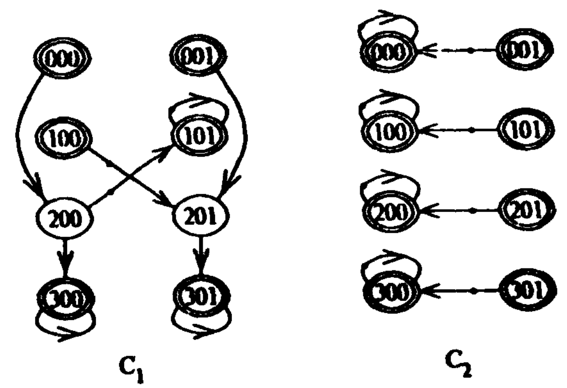


Figure B-1: First counterexample to Lemma 2

This example satisfies the conditions for Lemma 2, using Brand's definition of referring to Δ. It also satisfies requirement 1 above for both C_1 and C_2. In fact, it satisfies a stronger condition: Every state that is not a stopping has an ancestor and a descendant that are stopping points. But Lemma 2 fails for the sequence

$$100\ C_1\ 201\ C_2\ 200\ C_1\ 101 \quad .$$

This example does not satisfy the conditions for Lemma 2 using our stronger definition of referring to Δ, definition 1. Edges <200,300> and <201,301> in C_1 refer to Δ by this definition.

Figure B-2 shows edge sets C_1 and C_2 for the second counterexample. State sets D_1, D_2, and Δ are {0,1}. The stopping point sets are $S_1 = \{0,1\}$, $S_2 = \{0\}$, and $\Sigma = \{0\}$. Edges marked by a dot refer to Δ according to definition 1 (equivalent to Brand's definition for this example). The state name is xyd, where $x \in D_1$, $y \in D_2$, and $d \in \Delta$.

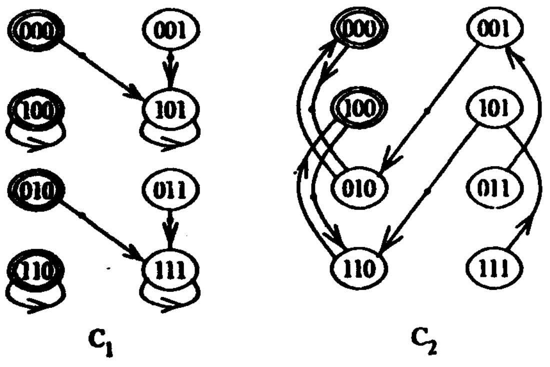


Figure B-2: Second counterexample to Lemma 2

Again, this example satisfies the conditions for Lemma 2. But Lemma 2 fails for the sequence

$$000\ C_1\ 101\ C_2\ 110\ C_2\ 100 \quad .$$

This example does not satisfy requirement 1. The paths $C_1{}^*$ from 000 and 010 do not reach stopping points.

B.2. Comments on Proof of Lemma 2

The proof is by induction. The proof for "no references to Δ" is trivially extended for "at most one reference to Δ". The inductive step is then for $(n+1) \geq 2$ references. After assuming the first reference to Δ is done by C_1, reorder the executions of C_1 and C_2, moving as many C_1 transitions as possible, as described in the proof. Then we have one of the three cases in Figure B-3.

Case I:

$$x = \langle a_0, b_0, d_0 \rangle\, C_1{}^*\, \langle a_1, b_0, d_0 \rangle\, \underline{C}_1\, \langle a_2, b_0, d_2 \rangle\, C_1{}^*\, \langle a_3, b_0, d_2 \rangle$$
$$\underline{C}_1\, \langle a_4, b_0, d_4 \rangle$$

Case II:

$$x = \langle a_0, b_0, d_0 \rangle\, C_1{}^*\, \langle a_1, b_0, d_0 \rangle\, \underline{C}_1\, \langle a_2, b_0, d_2 \rangle\, C_1{}^*\, \langle a_3, b_0, d_2 \rangle$$
$$\underline{C_2 \pm}\, \langle a_3, b_4, d_4 \rangle\, \underline{C}_1\, \langle a_5, b_4, d_5 \rangle$$

Case III:

$$x = \langle a_0, b_0, d_0 \rangle\, C_1{}^*\, \langle a_1, b_0, d_0 \rangle\, \underline{C}_1\, \langle a_2, b_0, d_2 \rangle\, C_1{}^*\, \langle a_3, b_0, d_2 \rangle$$
$$\underline{C_2 \pm}\, \langle a_3, b_4, d_4 \rangle = y$$

Figure B-3: Three cases in proof of Lemma 2

Figure B-3 shows the several possible types of state transition sequences, where the state space is $D_1 \times D_2 \times \Delta$. The transitions that refer to Δ are exactly those transitions for which the relation name is underlined. In the case of "$\underline{C_2}\pm$", more than one transition in the path may refer to Δ. States x and y are the stopping points of interest in the proof.

For each case, the proof asserts that one of the states in the path

$\langle a_2,b_0,d_2\rangle\, C_1{}^* \langle a_3,b_0,d_2\rangle$

is a stopping point. As shown by the counterexamples, this assertion fails without definition 1 and requirement 1. Now consider each case with the stronger requirements.

Case I: The presence of a stopping point in the path follows immediately from the requirement that two references to Δ in P_1 are separated by a stopping point.

Case II: Since program P_1 has no don't-cares, some edge h in C_1 is incident out of $\langle a_3,b_0,d_2\rangle$. Definition 1 implies that h refers to Δ, since edge

$\langle a_3,b_4,d_4\rangle\, C_1 \langle a_5,b_4,d_5\rangle$

refers to Δ. The addition of h to the path then makes this case like case I.

Case III: If $\langle a_2,b_0,d_2\rangle$ is a stopping point, we are done. Assume $\langle a_2,b_0,d_2\rangle$ is not a stopping point. Now $\langle a_2,b_0,d_2\rangle$ is reachable from stopping point x by $C_1{}^*$. Then, by requirement 1, every path $C_1{}^*$ from $\langle a_2,b_0,d_2\rangle$ reaches a stopping point. Let z be the first stopping point on one such path. This path $\langle a_2,b_0,d_2\rangle\, C_1{}^*\, z$ does not refer to Δ; otherwise two successive references to Δ by C_1 would not be separated by a stopping point, a violation of the condition assumed in the lemma. Therefore, $z^{\theta} = d_2$. Since z is a stopping point, we have $d_2 \in \Sigma$.

Since x and y are stopping points, we have $a_3 \in S_1$ and $b_0 \in S_2$. Therefore $\langle a_3,b_0,d_2\rangle$ is a stopping point.

Then, for each of the three cases, a stopping point is found on the path

$\langle a_2,b_0,d_2\rangle\, C_1{}^* \langle a_3,b_0,d_2\rangle$

as asserted in the proof of the lemma. This corrects the proof of Lemma 2.

Appendix C
Symbolic Execution of Blackjack Example

This appendix presents the results of the symbolic execution of the blackjack descriptions from Chapter 6.

C.1. Symbolic Execution with Case Splitting

This section presents the symbolic results, using case splitting, for the descriptions in Figures 6-1 and 6-2 as directed by the simulation relation R_{si}, defined by Figure 6-3. The results are obtained by the following procedure.

1. Select the initial location counters Lc_s for P_s (Figure 6-1) and Lc_i for P_i (Figure 6-2) corresponding to a pair of stopping points in R_{si}. In the Blackjack descriptions, Lc_i necessarily equals *top*, while Lc_s might be XB, for example.

2. Initialize the path condition Pc and the variables of P_s, as directed by R_{si} for states with the given location counters. For example, in the Blackjack machine, if we selected $Lc_s = XB$, then R_{si} implies $Hit_s = 1$, $Broke_s = 0$, and $Stand_s = 0$. (For Boolean variables, we will use $1 \equiv TRUE$ and $0 \equiv FALSE$.) In general, the initial path condition reflects requirements from R_{si} that cannot be inferred from the initial values given to variables. There are no such requirements in the Blackjack example.

In the Blackjack example, we will always have

- initial Pc equals $TRUE$,
- initial Hit_s, $Broke_s$, and $Stand_s$ have constant values, and
- initial FF_s, $Score_s$, $CardBuf_s$, $Value_s$, and $YCrd_s$ equal symbolic variables f_0, s_0, c_0, v_0, and y_0, respectively.

The only exception is when Lc_s equals XJ, in which case R_{si} implies $FF_s = 0$.

3. Execute P_s symbolically, introducing case splits if necessary, until P_s reaches another stopping point. For example, if Lc_s is XB initially, we symbolically execute P_s until Lc_s becomes XB (again) or XC.

4. Initialize the variables of P_i as directed by R_{si}. In the Blackjack example, we have

- Hit_i, $Broke_i$, $Stand_i$, FF_i, $Score_i$, $CardBuf_i$, $Value_i$, and $YCrd_i$ take the same initial values as their counterparts in P_s,
- initial $Q4_i$, $Q5_i$, and $Q6_i$ have constant values, and
- all remaining variables initially are set to distinct symbolic variables.

The final Pc from the execution of P_s becomes the initial Pc for this execution of P_i.

5. Execute P_i symbolically until Lc_i again equals *top*. In general, the execution of P_i could introduce more case splits or terminate with a different value of Lc_i, but that does not occur in the Blackjack example.

6. This completes the parallel execution of P_s (step (3)) and P_i (step (5)) for a single path in each description. Record the results.

7. If step (5) introduced case splits (this never occurs for the Blackjack example), then repeat steps (5) and (6) for each separate path in P_i.

8. If step (3) introduced case splits, then repeat steps (3) through (7) for each separate path in P_s.

9. Repeat steps (1) through (8) for each possible pair of starting location counters.

The results of the above procedure are presented below for all cases. To save space, we use the following conventions:

$State_s$ and $State_i$ are abbreviations for collections of variables. Initial values are specified only for $State_s$ and $State_i$. An initial value for *FF* also appears only if that value is not f_0. The initial values of *Score*, *CardBuf*, *Value*, and *YCrd* are always as given in step (2) above. The initial value of Lc_i is always *top*. The initial and final values of all remaining variables in P_i are always irrelevant and so are never listed.

= =

Results of symbolic executions of Figures 6-1 and 6-2 with case splitting

Figure 6-1 is description P_s

(* "@" denotes concatenation *)

$State_s \equiv Lc_s : Hit_s \,@\, Broke_s \,@\, Stand_s$

Figure 6-2 is description P_i

$State_i \equiv Hit_i \,@\, Broke_i \,@\, Stand_i \,@\, Q4_i \,@\, Q5_i \,@\, Q6_i$

= =

Pc: $TRUE$

Initial: $State_s = XA{:}000$

Final: $State_s = XB{:}100$ $FF_s=0$ $Score_s=0$ $CardBuf_s=c_0$

Initial: $State_i = 000000$

Final: $State_i = 100000$ $FF_i=0$ $Score_i=0$ $CardBuf_i=c_0$

= =

Pc: not y_0

Initial: $State_s = XB{:}100$

Final: $State_s = XB{:}100$ $FF_s=f_0$ $Score_s=s_0$ $CardBuf_s=v_0$

Initial: $State_i = 100000$

Final: $State_i = 100000$ $FF_i=f_0$ $Score_i=s_0$ $CardBuf_i=v_0$

= =

Pc: y_0

Initial: $State_s = XB{:}100$

Final: $State_s = XC{:}000$ $FF_s = f_0$ $Score_s = s_0$ $CardBuf_s = v_0$

- -

Initial: $State_i = 100000$

Final: $State_i = 000001$ $FF_i = f_0$ $Score_i = s_0$ $CardBuf_i = v_0$

= =

Pc: y_0

Initial: $State_s = XC{:}000$

Final: $State_s = XC{:}000$ $FF_s = f_0$ $Score_s = s_0$ $CardBuf_s = c_0$

- -

Initial: $State_i = 000001$

Final: $State_i = 000001$ $FF_i = f_0$ $Score_i = s_0$ $CardBuf_i = c_0$

= =

Pc: not y_0

Initial: $State_s = XC{:}000$

Final: $State_s = D{:}000$ $FF_s = f_0$ $Score_s = s_0$ $CardBuf_s = c_0$

- -

Initial: $State_i = 000001$

Final: $State_i = 000011$ $FF_i = f_0$ $Score_i = s_0$ $CardBuf_i = c_0$

= =

Pc: $(c_0 = 1)$ and not f_0

Initial: $State_s = D{:}000$

Final: $State_s = E{:}000$ $FF_s = 0$ $Score_s = s_0 + 1$ $CardBuf_s = 1$

values of FF_s and $CardBuf_s$ unchanged; implied by Pc

- -

Initial: $State_i = 000011$

Final: $State_i = 000111$ $FF_i = 0$ $Score_i = s_0 + 1$ $CardBuf_i = 1$

values of FF_i and $CardBuf_i$ unchanged; implied by Pc

= =

Pc: $(c_0 \neq 1)$ or f_0

Initial: $State_s = D{:}000$

Final: $State_s = F{:}000$ $FF_s = f_0$ $Score_s = s_0 + c_0$ $CardBuf_s = c_0$

- -

Initial: $State_i = 000011$

Final: $State_i = 000110$ $FF_i = f_0$ $Score_i = s_0 + c_0$ $CardBuf_i = c_0$

= =

Pc: $TRUE$

Initial: $State_s = E{:}000$

Final: $State_s = D{:}000$ $FF_s = 1$ $Score_s = s_0$ $CardBuf_s = 10$

- -

Initial: $State_i = 000111$

Final: $State_i = 000011$ $FF_i = 1$ $Score_i = s_0$ $CardBuf_i = 10$

must simplify (f_0 or not f_0) to get $FF_i = 1$

= =

Pc: $s_0 < 17$

Initial: $State_s = F{:}000$

Final: $State_s = XB{:}100$ $FF_s = f_0$ $Score_s = s_0$ $CardBuf_s = c_0$

Initial: $State_i = 000110$

Final: $State_i = 100000$ $FF_i = f_0$ $Score_i = s_0$ $CardBuf_i = c_0$

= =

Pc: $s_0 \geq 22$

Initial: $State_s = F{:}000$

Final: $State_s = H{:}000$ $FF_s = f_0$ $Score_s = s_0$ $CardBuf_s = c_0$

must optimize *Pc* from $(s_0 \geq 17)$ and $(s_0 \geq 22)$

Initial: $State_i = 000110$

Final: $State_i = 000100$ $FF_i = f_0$ $Score_i = s_0$ $CardBuf_i = c_0$

= =

Pc: $(s_0 \geq 17)$ and $(s_0 < 22)$

Initial: $State_s = F{:}000$

Final: $State_s = XK{:}001$ $FF_s = f_0$ $Score_s = s_0$ $CardBuf_s = c_0$

Initial: $State_i = 000110$

Final: $State_i = 001000$ $FF_i = f_0$ $Score_i = s_0$ $CardBuf_i = c_0$

= =

Pc: f_0

Initial: $State_s = H{:}000$

Final: $State_s = D{:}000$ $FF_s = 0$ $Score_s = s_0$ $CardBuf_s = -10$

Initial: $State_i = 000100$

Final: $State_i = 000011$ $FF_i = 0$ $Score_i = s_0$ $CardBuf_i = -10$

= =

Pc: not f_0

Initial: $State_s = H{:}000$

Final: $State_s = XJ{:}010$ $FF_s = 0$ $Score_s = s_0$ $CardBuf_s = -10$

value of FF_s unchanged; implied by *Pc*

Initial: $State_i = 000100$

Final: $State_i = 010000$ $FF_i = 0$ $Score_i = s_0$ $CardBuf_i = -10$

= =

Pc: y_0

Initial: $State_s = XJ{:}010$ $FF_s = 0$

Final: $State_s = XA{:}000$ $FF_s = 0$ $Score_s = s_0$ $CardBuf_s = c_0$

Initial: $State_i = 010000$ $FF_i = 0$

Final: $State_i = 000000$ $FF_i = 0$ $Score_i = s_0$ $CardBuf_i = c_0$

= =

Pc: **not** y_0

Initial: $State_s = XJ{:}010$ $FF_s = 0$

Final: $State_s = XJ{:}010$ $FF_s = 0$ $Score_s = s_0$ $CardBuf_s = c_0$

Initial: $State_i = 010000$ $FF_i = 0$

Final: $State_i = 010000$ $FF_i = 0$ $Score_i = s_0$ $CardBuf_i = c_0$

= =

Pc: y_0

Initial: $State_s = XK{:}001$

Final: $State_s = XA{:}000$ $FF_s = f_0$ $Score_s = s_0$ $CardBuf_s = c_0$

Initial: $State_i = 001000$

Final: $State_i = 000000$ $FF_i = f_0$ $Score_i = s_0$ $CardBuf_i = c_0$

= =

Pc: **not** y_0

Initial: $State_s = XK{:}001$

Final: $State_s = XK{:}001$ $FF_s = f_0$ $Score_s = s_0$ $CardBuf_s = c_0$

Initial: $State_i = 001000$

Final: $State_i = 001000$ $FF_i = f_0$ $Score_i = s_0$ $CardBuf_i = c_0$

= =

C.2. Symbolic Execution with Case Merging

This section presents the symbolic results, using case merging, for the descriptions in Figures 6-1 and 6-2. We obtain the results for Figure 6-1 by the following procedure:

1. Convert description P_s to a process of the form discussed in Chapter 5. Do this by inserting

   repeat pause until *TRUE*;

   just after each label that appears in the simulation relation R_{si}; that is, $XA, XB, XC, D, E, F, H, XJ$, and XK.
2. Generate a new process using the construction from Section 5.2.2. The pause loop in the new process will be

   repeat pause until *TRUE*;

   Note that the value of variable $LPc_s[i]$ corresponds to the predicate $(Lc_s = i)$.
3. Initialize the variables in the new process as specified under "Initial:" below for description P_s. Variable p_0 is a symbolic variable representing the location counter in the initial state. The initial values of elements of LPc are expressions referring to p_0. The initial values of the remaining variables are distinct symbolic variables.
4. Symbolically execute the main body of the new process.

Steps (3) and (4) are similar to the symbolic execution mentioned in the second paragraph of Section 5.2.3. Note that this procedure does not make use of R_{si}. In the results given below, we assume the use of a dataflow analysis to make easy simplifications to the symbolic results, but the results are not otherwise simplified or put in a normal form.

Figure 6-2 contains no conditional branching, so the above procedure simplifies to the following:

1. Initialize the variables as given below for description P_i. We use many of the same symbolic variables used to initialize P_s, in agreement with R_{si}.
2. Symbolically execute the loop once.

The results for the two descriptions follow.

= =

Results of symbolic executions of Figures 6-1 and 6-2 with case merging

$State_i \equiv Hit_i \ @ \ Broke_i \ @ \ Stand_i \ @ \ Q4_i \ @ \ Q5_i \ @ \ Q6_i$

= =

Initial:

$\forall i \notin \{XA,XB,XC,D,E,F,H,XJ,XK\}$: $LPc_s[i] = FALSE$

$\forall i \in \{XA,XB,XC,D,E,F,H,XJ,XK\}$: $LPc_s[i] = (p_0 = i)$

$Hit_s = h_0$	$Broke_s = b_0$	$Stand_s = t_0$
$FF_s = f_0$	$Score_s = s_0$	$CardBuf_s = c_0$
$YCrd_s = y_0$	$Value_s = v_0$	

Final:

$\forall i \notin \{XA,XB,XC,D,E,F,H,XJ,XK\}$: $LPc_s[i] = FALSE$

$LPc_s[XA] = ((p_0 = XJ) \text{ and } y_0) \text{ or } ((p_0 = XK) \text{ and } y_0)$

$LPc_s[XB] = (p_0 = XA) \text{ or } ((p_0 = XB) \text{ and not } y_0) \text{ or }$
$\quad ((p_0 = F) \text{ and } (s_0 < 17))$

$LPc_s[XC] = ((p_0 = XB) \text{ and } y_0) \text{ or } ((p_0 = XC) \text{ and } y_0)$

$LPc_s[D] = ((p_0 = XC) \text{ and not } y_0) \text{ or } (p_0 = E) \text{ or } ((p_0 = H) \text{ and } f_0)$

$LPc_s[E] = (p_0 = D) \text{ and } (c_0 = 1) \text{ and not } f_0$

$LPc_s[F] = (p_0 = D) \text{ and } ((c_0 \neq 1) \text{ or } f_0)$

$LPc_s[H] = (p_0 = F) \text{ and } (s_0 \geq 17) \text{ and } (s_0 \geq 22)$

$LPc_s[XJ] = ((p_0 = H) \text{ and not } f_0) \text{ or } ((p_0 = XJ) \text{ and not } y_0)$

$LPc_s[XK] = ((p_0 = F) \text{ and } (s_0 \geq 17) \text{ and } (s_0 < 22)) \text{ or }$
$\quad ((p_0 = XK) \text{ and not } y_0)$

$Hit_s = (p_0 = XA) \text{ or } ((p_0 = XB) \text{ and not } y_0) \text{ or } ((p_0 = F) \text{ and } (s_0 < 17)) \text{ or }$
$\quad (h_0 \text{ and not } (y_0 \text{ and } p_0 \in \{XB,XC,XJ,XK\}))$

$Broke_s = (((p_0 = H) \text{ and not } f_0) \text{ or } ((p_0 = XJ) \text{ and not } y_0)) \text{ or }$
$\quad (b_0 \text{ and not } (y_0 \text{ and } p_0 \in \{XJ,XK\}))$

$Stand_s = (((p_0 = F) \text{ and } (s_0 \geq 17) \text{ and } (s_0 < 22)) \text{ or } ((p_0 = XK) \text{ and not } y_0)) \text{ or }$
$\quad (t_0 \text{ and not } (y_0 \text{ and } p_0 \in \{XJ,XK\}))$

$FF_s = (p_0 = E)$ or $(f_0$ and $p_0 \in \{XB, XC, D, F, XJ, XK\})$

$Score_s =$ if $p_0 = XA$ then 0 else

  if $p_0 = D$ then $s_0 + c_0$ else

  if $p_0 \notin \{XA, D\}$ then s_0

$CardBuf_s =$ if $p_0 = XB$ then v_0 else

  if $p_0 = E$ then 10 else

  if $p_0 = H$ then -10 else

  if $p_0 \notin \{XB, E, H\}$ then c_0

Initial:

$State_i = h_0 b_0 t_0 u_0 w_0 x_0$

$FF_i = f_0$	$Score_i = s_0$	$CardBuf_i = c_0$
$YCrd_i = y_0$	$Value_i = v_0$	

Final:

$Hit_i = (($not $(b_0$ or t_0 or u_0 or $x_0)$ or

  w_0 and not x_0 and $(s_0 < 17))$ and not $h_0)$ or

  not y_0 and h_0

$Broke_i = (u_0$ and not w_0 and not f_0 and not $b_0)$ or

  not y_0 and b_0

$Stand_i = (w_0$ and not x_0 and $(17 \leq s_0 < 22)$ and not $t_0)$ or

  not y_0 and t_0

$Q4_i = w_0$ and not u_0 or

  not (not w_0 or x_0 or $(s_0 < 22))$ and u_0

$Q5_i = (x_0$ and not y_0 or u_0 and $f_0)$ and not w_0 or

  not $(u_0$ and not $x_0)$ and w_0

$Q6_i = (h_0$ and y_0 or u_0 and not w_0 and $f_0)$ and not x_0 or

  not $(((c_0 \neq 1)$ or $f_0)$ and not u_0 and $w_0)$ and x_0

$FF_i = (u_0$ and x_0 and not $f_0)$ or

  $(h_0$ or t_0 or w_0 or $x_0)$ and f_0

$Score_i =$ if not $(h_0$ or b_0 or t_0 or u_0 or $x_0)$ then 0 else

  if (not u_0 and w_0) then $s_0 + c_0$ else s_0

$CardBuf_i =$ if h_0 then v_0 else

  if $(u_0$ and $x_0)$ then 10 else

  if $(u_0$ and not $w_0)$ then -10 else c_0

===================================

References

[AH74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

[AU77] A.V. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1977.

[BB79] M.R. Barbacci, G.E. Barnes, R.G. Cattell, and D.P. Siewiorek. *The ISPS Computer Description Language.* Report CMU-CS-79-137, Carnegie-Mellon University, Department of Computer Science, August 1979.

[BF76] M.A. Breuer and A.D. Friedman. *Diagnosis & Reliable Design of Digital Systems.* Computer Science Press, Potomac, Maryland, 1976.

[Bo82] G.V. Bochmann. Hardware Specification with Temporal Logic: An Example. *IEEE Trans. Computers* C-31(3):223-231, March 1982.

[Br78] D. Brand. *Algebraic Simulation between Parallel Programs.* Research Report RC 7206, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., June 1978. 39 pages.

[CE77] W.C. Carter, H.A. Ellozy, W.H. Joyner Jr., and G.B. Leeman Jr. *Techniques for Microprogram Validation.* Research Report RC 6361, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., January 1977. 19 pages.

[CH79] T.E. Cheatham, Jr., G.H. Holloway, and J.A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Trans. Software Engineering* SE-5(4):402-417, July 1979.

[CJ78] W.C. Carter, W.H. Joyner Jr., and D. Brand. Microprogram Verification Considered Necessary. In *Proc. National Computer Conference*, pages 657-664. AFIPS, 1978.

[CJ79] W.C. Carter, W.H. Joyner Jr., and D. Brand. Symbolic Simulation for Correct Machine Design. In *Proc. 16th Design Automation Conference*, pages 280-286. IEEE and ACM, San Diego, 1979.

[CM80] S.D. Crocker, L. Marcus, and D. van-Mierop. The ISI Microcode Verification System. In *Proc. IFIP Working Conference on Firmware, Microprogramming and Restructurable Hardware*, pages 89-103. International Federation for Information Processing, Linz, Austria, April 1980.

[Co81] W.E. Cory. Symbolic Simulation for Functional Verification with ADLIB and SDL. In *Proc. 18th Design Automation Conference*, pages 82-89. IEEE and ACM, Nashville, June 1981.

[Cv82] W.E. Cory and W.M. vanCleemput. Developments in Verification of Design Correctness--A Tutorial. In Rex Rice (editor), *Tutorial--VLSI Support Technologies: Computer-Aided Design, Testing, and Packaging*, pages 169-177. IEEE Computer Society Press, Los Alamitos, CA, 1982. Reprinted from *Proc. 17th Design Automation Conference*, pages 156-164, IEEE and ACM, Minneapolis, June, 1980.

[Da79] J.A. Darringer. The Application of Program Verification Techniques to Hardware Verification. In *Proc. 16th Design Automation Conference*, pages 375-381. IEEE and ACM, San Diego, June 1979.

[DD75] D.L. Dietmeyer and J.R. Duley. Register Transfer Languages and Their Translation. In M.A. Breuer (editor), *Digital System Design Automation: Languages, Simulation & Data Base*, chapter 2. Computer Science Press, Woodland Hills, CA, 1975. pages 117-218.

[EG76] C.J. Evangelisti, G. Goertzel, and H. Ofek. *LCD--Language for Computer Design (Revised).* Research Report RC 6244, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., October 1976. 32 pages.

[EG77] C.J. Evangelisti, G. Goertzel, and H. Ofek. Designing with LCD--Language for Computer Design. In *Proc. Fourteenth Design Automation Conference*, pages 369-376. IEEE and ACM, New Orleans, 1977.

[Fl67] R.W. Floyd. Assigning Meanings to Programs. In *Proc. Symposia in Applied Mathematics*, pages 19-32. American Mathematical Society, 1967.

[HC75] S.L. Hantler and A.C. Chibib. *EFFIGY Reference Manual.* Research Report RC 5225, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., January 1975.

[Hi79] D.D. Hill. ADLIB: A Modular, Strongly-Typed Computer Design Language. In *Proc. Fourth Intl. Symposium on Computer Hardware Description Languages*, pages 75-81. IEEE and ACM, Palo Alto, 1979.

[Hi80] D.D. Hill. *Language and Environment for Multi-Level Simulation.* PhD thesis, Stanford University, Dept. of Electrical Engineering, March 1980. also Technical Report No. 185, 170 pages.

[HK76] S.L. Hantler and J.C. King. An Introduction to Proving the Correctness of Programs. *ACM Computing Surveys* 8(3):331-353, September 1976.

[Ho69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM* 12(10):576-580,583, October 1969.

[HR74] H. Halliwell and J.P. Roth. SCD A System for Computer Design. *IBM Technical Disclosure Bulletin* 17(5):1517-1519, October 1974.

[Hv79] D.D. Hill and W.M. vanCleemput. SABLE: A Tool for Generating Structured, Multi-level Simulations. In *Proc. Sixteenth Design Automation Conference*, pages 272-279. IEEE and ACM, San Diego, June 1979.

[JC76] W.H. Joyner, Jr., W.C. Carter, G.B. Leeman, Jr. Automated Proofs of Microprogram Correctness. In *Proc. Ninth Annual Workshop on Microprogramming*, pages 51-55. IEEE and ACM, New Orleans, September 1976.

[JW79] K. Jensen and N. Wirth. *Pascal User Manual and Report, Second Edition.* Springer-Verlag, New York, 1979.

[Ki75] J.C. King. A New Approach to Program Testing. *ACM SIGPLAN Notices* 10(6):228-233, June 1975. This issue is *Proc. 1975 International Conference on Reliable Software*, Los Angeles, April 1975.

[Ki76a] J.C. King. Symbolic Execution and Program Testing. *Communications of the ACM* 19(7):385-394, July 1976.

[Ki76b] J.C. King. On Generating Verification Conditions for Correctness Proofs. In *4. Fachtagung der GI Programmiersprachen*, pages 253-267. Gesellschaft Fur Informatik, Erlangen, Germany, March 1976. Published by Springer-Verlag: New York; paper is in English. Also available as Research Report RC 5808, January 1976, from IBM T.J. Watson Research Center, Yorktown Heights, N.Y.

[KS79] N. Kawato, T. Saito, F. Maruyama, and T. Uehara. Design and Verification of Large-Scale Computers by Using DDL. In *Proc. 16th Design Automation Conference*, pages 360-366. IEEE and ACM, San Diego, 1979.

[LG79] D.C. Luckham, S.M. German, F.W. v.Henke, R.A. Karp, P.W. Milne, D.C. Oppen, W. Polak, and W.L. Scherlis. *Stanford Pascal Verifier User Manual.* Report No. STAN-CS-79-731, Stanford University, Computer Science Department, March 1979. 115 pages.

[LJ77] G.B. Leeman Jr., W.H. Joyner Jr., and W.C. Carter. *An Automated Proof of Microprogram Correctness.* Research Report RC 6587, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., June 1977. 33 pages.

[LL79] S. Leinwand and T. Lamdan. Design Verification Based on Functional Abstraction. In *Proc. 16th Design Automation Conference*, pages 353-359. IEEE and ACM, San Diego, 1979.

[LS79] D.C. Luckham and N. Suzuki. Verification of Array, Record, and Pointer Operations in Pascal. *ACM Trans. Programming Languages and Systems* 1(2):226-244, October 1979.

[Ma66] P. Martin-lof. The Definition of Random Sequences. *Information and Control* 9(6):602-619, December 1966.

[Mc62] J. McCarthy. Towards a Mathematical Science of Computation. In *Proc. IFIP Congress 62*, pages 21-28. IFIP, Munich, August 1962.

[Mi71] R. Milner. An Algebraic Definition of Simulation between Programs. In *Proc. Second International Joint Conference on Artificial Intelligence*, pages 481-489. British Computer Society, London, September 1971.

[MP81] Z. Manna and A. Pnueli. *Verification of Concurrent Programs, Part I: The Temporal Framework.* Report No. STAN-CS-81-836, Stanford University, Computer Science Department, June 1981. 62 pages.

[OL78] H. Ofek, J.D. Lesser, C.J. Evangelisti, and G. Goertzel. *Structured Design Verification of Sequential Machines.* Research Report RC 7037, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., March 1978. 20 pages.

[Pa76] D.A. Patterson. STRUM: Structured Microprogramming System for Correct Firmware. *IEEE Trans. Computers Special Issue on Microprogramming* , October 1976.

[PS82] V. Pitchumani and E.P. Stabler. A Formal Method for Computer Design Verification. In *Proc. 19th Design Automation Conference*, pages 809-814. IEEE and ACM, Las Vegas, June 1982.

[Ro66] J.P. Roth. Diagnosis of Automata Failures: A Calculus and a Method. *IBM Journal of Research and Development* 10:278-291, July 1966.

[Ro73] J.P. Roth. Verify: An Algorithm to Verify a Computer Design. *IBM Technical Disclosure Bulletin* 15(8):2646-2648, January 1973.

[Ro75] J.P. Roth. *Generation and Verification of Hardware Designs at High Level.* Research Report RC 5779, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., November 1975. 16 pages.

[Ro77] J.P. Roth. Hardware Verification. *IEEE Trans. Computers* C-26(12):1292-1294, December 1977.

[Ul65] E.G. Ulrich. Time-Sequenced Logical Simulation Based on Circuit Delay and Selective Tracing of Active Network Paths. In *Proc. 20th National Conference*, pages 437-448. ACM, Cleveland, August 1965.

[Un81] S.H. Unger. Double-Edge-Triggered Flip-Flops. *IEEE Trans. Computers* C-30(6):447-451, June 1981.

[va77] W.M. vanCleemput. *A Structural Design Language for Computer Aided Design of Digital Systems.* Technical Report 136, Stanford University, Computer Systems Laboratory, April 1977. 27 pages.

[va79] D. van-Mierop. *An Experiment in Automatic Verification of Microcode.* Microver note #31, University of Southern California, Information Sciences Institute, June 1979. 21 pages.

[vM78] D. van-Mierop, L. Marcus, and S.D. Crocker. Verification of the FTSC Microprogram. In *Proc. 11th Annual Microprogramming Workshop*, pages 118. IEEE and ACM, Pacific Grove, CA, November 1978.

2

DTI